A Final Report

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Submitted by:

John C. Knight

Samuel T. Gregory

John I. A. Urquhart

Report No. UVA/528213/AMCS84/104

February 1984

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements

Final Report

John C. Knight
Samuel T. Gregory
John I. A. Urquhart

Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville
Virginia, 22901

February 1984

CONTENTS

1. INTRODUCTION

The purpose of this grant is to investigate the use and implementation of Ada (a trade mark of the US Dept. of Defense) in distributed environments in which the hardware components are assumed to be unreliable. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

During the twelve-month period covered by this grant, our primary activities have been designing our fault-tolerant Ada System and implementing an initial version of it. We have completed much of the design work although some details remain to be worked out. The design has been influenced by our desire to use this system for experimentation. Many features have been included to facilitate analysis and verify results. Consequently we have chosen to ignore efficient execution and to stress simplicity and flexibility.

A first version of the implementation has also been completed and is being tested. At present all the major features of the tasking and exception mechanisms of Ada have been implemented. The interface that the testbed presents to the user is still extremely crude and is presently the subject of revision. Our original intention was to implement only an execution-time system for Ada and not to bother building a translator. We felt that whatever programs were needed for demonstration or experiment could be compiled by hand. We discovered how wrong this was when the hand compilation of the first test program took a whole day. Consequently, we have begun the development of a translator for the subset of Ada that pertains to this research.

In Section 2 of this report, the hardware configurations we are using and intend to use are described. An overview of the system design is presented in Section 3. A brief description of the translator mentioned above and its present status is given in Section 4.

We have continued our analysis of the Ada language and found new and potentially serious difficulties with task termination. These issues are described specifically in Section 5. Two papers have been prepared that discuss the difficulties with Ada. One of them was presented at the AIAA Conference On Computers And Aerospace held in Hartford in October, 1983. A copy of that paper is included in this report as Appendix 2. The second paper is a still in draft form and will be submitted to a journal when complete. It is included in this report as Appendix 5. The reader is cautioned that Appendix 5 will not be the final version of that paper, and that inevitably there is some overlap between that paper and the one in Appendix 2.

A consequence of our analysis of Ada is a set of general impressions about what features are needed in languages for fault-tolerant distributed processing. We have documented these impressions in a paper that has been submitted to the Fourteenth Annual Fault-Tolerant Systems Conference. A copy of that paper is included in this report as Appendix 3. We have not received the decision of the conference program committee about that paper.

Our distributed Ada implementation will constitute an incomplete, but useful, operational semantic definition of Ada tasking. The purpose of a semantic definition is to answer questions of language meaning. An operational definition does this by allowing programs to be executed and their actions to be observed.

Existing operational semantic definitions such as NYU's Ada/ED are not adequate for concurrent languages. The reason is that a "what if" question about tasking cannot be answered because a set of tasks cannot be forced into the necessary state which leads to the "what if" question. Note that this does not happen with sequential languages because they are deterministic. Concurrent languages are non-deterministic and this means that there may be no guarantee that a particular state of interest is reached on any given execution.

To avoid this problem we have designed a sequence control system which will allow the progress of individual tasks to be adjusted so that a program can be forced into any particular state. This is discussed in depth in Section 6. The sequence control mechanism is a part of the testbed and a paper describing the testbed has been submitted to the Fourteenth Annual Fault-Tolerant Systems Conference. A copy of that paper is included in this report as Appendix 4. We have not received the decision of the conference program committee about that paper either.

Our work on Ada has attracted attention from outside the University of Virginia. During the grant reporting period seminars have been given describing the work at:

(1) The Research Triangle Institute (twice).

(2) The University of North Carolina at Chapel Hill.

(3) North Carolina State University.

(4) IBM Federal Systems Division.

A presentation describing our work was given at a conference on the general topic of fault tolerance organized by General Electric Corporation. The conference was held in Charlottesville and attended mainly by GE personnel from various locations.

2. HARDWARE CONFIGURATIONS

One of the criticisms frequently made of demonstrations is that asynchronous hardware is often simulated and, consequently, result obtained cannot be relied upon. Since we are attempting to demonstrate reliability of a system involving several computers, we will be using a hardware configuration with several computers in it. However, for the purposes of software development we will be using a simulation of a multicomputer configuration running on a DEC VAX 11/780. Both of these configurations are described in this section.

2.1. Physical Multicomputers

We have purchased two IBM Personal Computers (PCs) to be used in conjunction with three other PCs owned by another research project. We will also be using the department's DEC VAX 11/780 computer. The PCs will be connected to the VAX by low-speed serial lines and these lines will allow software and data files developed on the VAX to be transferred to the PCs. Our original intention was to route all PC-to-PC communication through the VAX using these serial lines. This would have allowed the VAX to monitor all PC-to-PC communication and to control the PCs by sending messages to them which it originated. This mechanism would have been used for debugging, and for initiating and monitoring reconfiguration experiments. Careful review of this plan showed that it would be very difficult to implement. The reason is that since all messages would be manipulated by both Pcs and the VAX, it would be necessary to have bit-level access to message formats on both

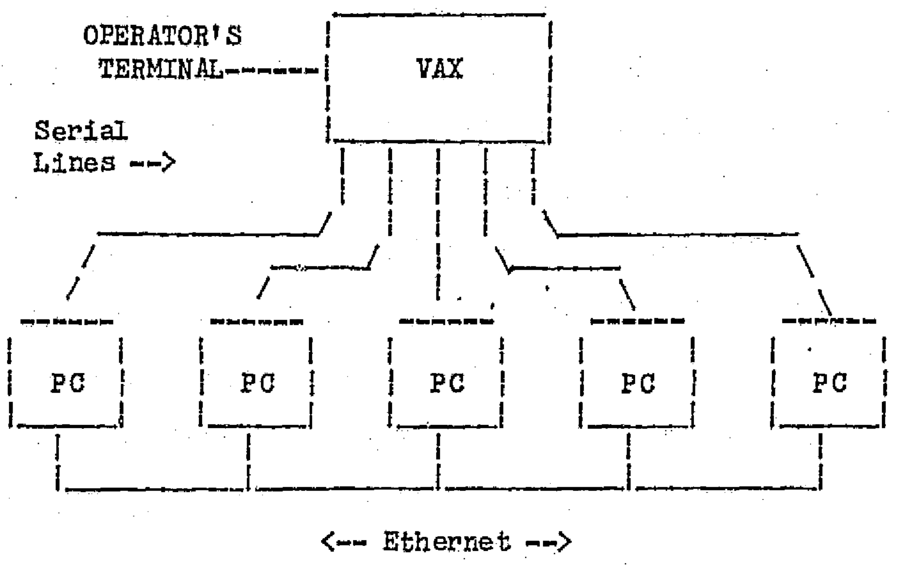


Figure 1 - Hardware Configuration.

computers. All of our software is implemented in Pascal and all messages are defined as Pascal records. Since VAX Pascal and IBM PC Pascal implement records differently, this would make symbolic access to the messages on both machines extremely difficult. This is such a substantial problem that we chose to abandon the approach.

The PCs are also connected by a high-speed Ethernet system that is not routed through the VAX. The necessary Ethernet hardware for the two PCs purchased under this grant has been provided by the University of Virginia's Department Of Applied Mathematics and Computer Science. The equipment was manufactured by TECMAR Inc.

A major part of this grant period has been spent in implementing the software necessary to allow the PCs to communicate using the Ethernet hardware. Although the equipment was manufactured by TECMAR specifically for the IBM Personal Computer, it leaves a great deal to be desired. The hardware is poorly designed. For example, only a single buffer is provided so that, in principle, each hardware unit can only be used for transmission or reception, not both, at any given time. We have circumvented this problem in software at the cost of some loss of performance. The documentation provided by TECMAR is extremely poor. Not only is it incomplete but it contains numerous errors in the detailed description of how the hardware works. Consequently very substantial delays were incurred by relying on the documentation and not understanding why the system would not work correctly.

The necessary software to support our testbed has now been completed and tested. It works to our satisfaction, and, in our opinion, is far better than any software available form TECMAR Inc.

The proposed configuration is shown in Figure 1.

2.2. Logical Multicomputers

We feel that the IBM PC does not represent a suitable environment for developing the software necessary for this project. As a consequence of the Ada semantics, much of the software is very complex and developing it requires powerful support facilities such as those provided by UNIX.

We have been using UNIX to develop our testbed software and we felt it was important that it be possible to test the software under UNIX. Consequently, we have put substantial effort into constructing a software analogy of the IBM PC/Ethernet configuration. This analogy uses UNIX processes to simulate IBM PC processors, UNIX pipes to simulate the Ethernet communications facility, and UNIX terminals to simulate the monitors and keyboards of the PCs. Thus we are able to execute the testbed on the VAX in an asynchronous environment that is reasonably realistic. This mechanism is described in more detail below. In the remainder of this report, any capability described as running on the IBM PCs will also run on the UNIX process/pipe implementation.

3. SOFTWARE SYSTEM DESIGN

The software system is in two parts. One part, called the sequencer, runs on one PC and the other, called the interpreter, on the remaining PCs (one copy on each).

The sequencer controls the entire system. It communicates with the experimenter via an interactive terminal and processes a command language. Commands are then implemented by sending special purpose messages to the remaining PCs. These commands allow programs to be loaded, parts of the system to be deliberately failed, and so on. The sequencer also implements the sequence control system (See below).

The software running on the remaining PCs actually executes the distributed Ada program. We have chosen not to generate native Intel 8088 code for the PCs but to generate code for a synthetic machine which will be interpreted. Our reasons are:

(1) It would be difficult to retain complete control of the program if the PCs were executing it directly.

(2) Generating code for a synthetic machine will make code generation very much simpler.

(3) The synthetic machine architecture can itself be the subject of experimentation. This will allow investigation of hardware designs which can support distribution.

(4) The software can be moved to different physical computers very easily.

The PC software is organized as three major layers. The first layer provides communication facilities. It accepts and delivers complete messages from the rest of the system and interfaces at the character level with the serial line and the Ethernet. Since the knowledge of the two communications lines is hidden in this layer, it will be relatively simple to use either as desired.

Although we will have up to five IBM PCs available for experimentation, we find it desirable to be able to model arbitrary distributed systems. We have defined the concept of an "abstract processor" (AP) that is a generic processor suitable for use as a general node in a distributed system. A set of abstract processors will be the distributed system that is presented to the Ada program.

Each PC will implement an arbitrary number of APs. This is the function of the second layer of the PC software. Thus a distributed system comprising a set of any number of processors could be run on any number of PCs; from one to five. In addition, since the system can be run on the VAX, a single physical processor can appear to be any desired distributed configuration.

This use of the abstract processors will allow experimentation with any desired distributed configuration. For demonstration purposes however, each PC can be made to run exactly one AP and the logical system will then be equivalent to the physical system.

Since an Ada program can consist of any number of tasks, and tasks can be created dynamically, each AP in a distributed system must be able to support any number of Ada tasks. The usual approach to the implementation of multiple tasks on a single processor is to multiplex the real processor and give each task the impression that it has its own, rather slow, processor. Ada tasks will be implemented this way and each AP will support an arbitrary number of virtual processors (VPs) with one for each Ada task. The provision of the VPs is the function of the third layer of PC software.

The virtual processors are designed to make Ada task execution fairly easy. Their instruction sets are tailored to Ada tasking and they have special "hardware" features such as built-in entry queues. The software therefore does not have to implement these queues.

The virtual processors must support the entire semantics of Ada tasks and so their implementation is quite complicated. As we discover more about the language semantics so the complexity of the VPs increases. The "hardware reference manual" for the virtual processors is included in this report as Appendix 1.

All of the processors in the system communicate via a set of messages. Thus for example a rendezvous is implemented as a series of messages even when the VPs involved are executing on the same AP. Some messages (those between APs on a single PC) do not get transmitted through the Ethernet. The PC communications software reflects or "mirrors" these messages back to the appropriate AP.

In our original discussion of fault detection, we proposed a system of software heartbeats which would allow detection of failed equipment. In our design, we have included the heartbeat mechanism at the higher levels of abstraction, but we will not include the heartbeats in our initial implementation. The reason is that they add nothing to the experiments that we wish to perform. We are concerned with the events occurring following a fault and these are best observed if the fault is deliberately injected. This will be done from the command language supported by the sequencer software. Failure of an AP will be communicated to the PC responsible for running that AP by a message. The AP will then cease being scheduled by the PC. The remainder of the APs will be informed by a broadcast message. Thus instead of the heartbeat mechanism, the experimenter will be able to cause any of the APs to fail at any desired point. The heartbeat mechanism will be added when the rest of the system is complete and running smoothly.

4. Ada SUBSET TRANSLATOR

In order to allow us to debug our testbed and to perform experiments we need to be able to execute a variety of Ada programs. Preparing these programs for execution requires a translator and a major portion of the grant reporting period has been spent designing and implementing this translator. It is important to understand that the target of this translator is the virtual processors described above and not the IBM PC or the DEC VAX 11/780.

Our goal in this project does not include compiler research and so we sought the most timely manner of producing the translator we needed rather than spending a lot of time developing fast, efficient, or otherwise noteworthy compilation techniques. Consequently, our approach has been to modify an existing translator for a language that was in some ways similar to Ada. This translator had originally been built using the UNIX compiler construction tools - YACC and LEX. The translator is written in C.

The translator for our subset of Ada is essentially complete and generates code for the testbed's virtual processors. The translator is presently being tested.

5. LANGUAGE ISSUES

As a part of designing the software for the VP software layer we had to look closely at the semantics of task termination. This had been discussed on several occasions and thought to be well understood. The current discussions centered around efficient methods of implementation. The most efficient, and we believe the usual, method of handling termination is for a master task to count the number of its dependents who are ready to terminate, and to terminate the group when the number counted equals the number of dependents. The count may decrease as well as increase because a task may become "unready" if one of its entries is called by another task.

On a uniprocessor this is not a problem and yields a valid implementation. On a distributed system it may not. The Ada Language Reference Manual (LRM) states that a necessary and sufficient condition for termination is:

> "Each task that depends on the master _is_ either already terminated or similarly waiting on an open terminate alternative of a select statement".

In a distributed system, determination of a task's state on a remote machine has to be determined by message passing. The above condition uses the present tense and therefore a task may not change its state once asked about termination until the master has made its decision to terminate or not. This means that if termination is not possible, the master must send a second message indicating that a task may resume execution.

There are two things to note here. The first is that failure of the processor running the master between the two required messages will suspend the dependent task permanently. This is similar to many other conditions we have noted before. The second thing to note is that counting dependents is not a valid implementation because it records task states as they were. Termination has to be based on a "snapshot" of task states.

The reason that this is not a major problem on a uniprocessor is that a snapshot can trivially be obtained by a master since while it is executing, all its dependents are suspended.

The Ada text shown in figure 2 is a set of tasks which should never terminate. Despite the TERMINATE alternates in the SELECT statements, any implementation which terminates these tasks is wrong. Task X is unable to terminate since tasks A and B are in an infinite loop of alternating instigations of rendezvous. Any termination-check algorithm which does not stop both A and B, such as a dependent counting algorithm, allows the possibility for the combination of old and new information to indicate that BOTH A and B are waiting at select statements with open terminate alternatives, which is clearly impossible.

Also note that, with an algorithm which stops all dependents periodically for polling, task X will actually be interfering with the progress of tasks A and B throughout their lives.

We feel that we understand the language issues involved with Ada operating on distributed systems at this point. Papers describing the

```
procedure DEMO is

   task X;

   task body X is

      task A is entry E; end A;

      task B is entry E; end B;

      task body A is begin
         loop
            B.E;
            select
               accept E;
            or
               terminate;
            end select;
         end loop;
      end A;

      task body B is begin
         loop
            select
               accept E;
            or
               terminate;
            end select;
            A.E;
         end loop;
      end B;

   begin null; end X;

begin null; end DEMO;
```

Figure 2 - Tasks Which Should Never Terminate

different aspects of this are included in this report in Appendix 2, Appendix 3, and Appendix 5.

6. SEQUENCER

The sequence of actions within the testbed will be controlled from the sequencer. In most simulators, there is a single step mode whereby effects of individual instructions within a program can be studied in detail. The case of simulating parallel programs, particularly when distributed over many machines, is more complicated. Not only is it necessary to single step individual tasks, but is it also necessary to single step them in relation to each other. Further, our interests in this research are such that we want to deal with the tasks through a perspective which is more microscopic than the Ada source language level. A typical experiment is expected to arrange for an accepting task to send a CHECK_CALLER message just after the calling task's timed entry call times out. Both tasks must be held at points within Ada statements so as to force the required interaction.

The sequencer is a part of the testbed software and its purpose is to control the parallel activities of the tasks within an Ada program. It deals with the program at the interpreted intermediate code level. It provides breakpoints and allows tasks to be single stepped in terms of individual messages as well as the interpreted instructions.

A scheme has been worked out by which an actual distributed system could establish communication and start up without the sequencer. However, in order for the sequencer to maintain control, it must establish its control at start-up, thus it handles the assignment of APs to PCs and PC names to ports.

The sequencer is interactive and is able to display all relevant tables owned by the PCs and by the message switch. It also permits a fast-forward interpretation mode to allow the system to set up the desired experiment without requiring a great deal of the experimenter's time. Since code is shared among tasks, breakpoints by code location only are insufficient. A breakpoint is named by source-level task name (task id), code location, and number of times the task must execute that code location before "hitting" the breakpoint. This count is due to the experimenter's desire to perform experiments within loops or at (trailing) end conditions. Implications of the source level task id are that:

(1) There will be no more than one (1) execution of allocators of tasks for any access variable.

(2) There will be no two identical simple names for tasks within any experimental program.

These are not serious restrictions.

As part of its control of the PCs, the sequencer must have extensive communications with them. All of these messages are copied into a log file to allow for later detailed study. Further, in order to direct individual task's (i.e. VP's) actions, the sequencer must also maintain copies of all the AP's VP maps.

Brief descriptions of the sequencer commands are:

(1) Load AP to PC map information into both the sequencer and the individual PCs.

(2) Load program to be run. The compiler's output has been stored in a file on the VAX and the appropriate file for this experiment is copied into the individual PCs' memories.

(3) Inform all PCs of the demise of a particular AP. This causes the PC owning the subject AP to cease to schedule it.

(4) Allow the named tasks to run until they encounter breakpoints or terminate. The absence of task names indicates that all tasks should be run (none should be artificially suspended).

(5) Stop or artificially suspend the named tasks no matter what they are doing. Absence of parameters means stop all tasks.

(6) Single step the named task through the interpretation of one instruction.

(7) Single step the named task through the handling of one message. This has been separated from the command to single step an instruction to allow better control of the order of events within a VP.

(8) Display the sequencer's tables. Due to table vs. terminal screen size, these may have to be individually selected. The tables needed by the sequencer are task_id to VP_name, VP_name to AP_number, AP_number to PP_number, PP_number to port, and relevant breakpoints.

(9) Remove breakpoint. Inverse operation of the set breakpoint command.

Part of the sequencer's monitoring interface provided for the experimenter is actually provided by the individual PCs on their own terminal screens. This is a set of displays selectable at the PCs' keyboards. The displays are a summary with one line per VP including minimal status information and any simulated output control signals, full VP status for one VP and utilizing the entire screen, full AP status for one AP, and full status of the PC itself. These status reports include contents of message buffers as well as internal tables.

APPENDIX 1

Ada* Virtual Processor Hardware Manual

Samuel T. Gregory

*Ada is a trademark of the US DoD (AJPO)

This is a description of the testbed's virtual processors. Names taken from the Pascal-like declarations below and appearing in the text are delimited by the character "!".

Certain sample instruction sequences are given toward the end of this document as aids in the compiler's author's task. As a further aid, two simple Ada programs are shown along with their translations to instruction sequences and template sequences.

The thread of control associated with an Ada task (including the environment task) represents the execution of exactly one target machine.

The target machine has the following kinds of memories:

(1) A static memory for instructions and string constants.

(2) A static memory for templates.

(3) A set of entry queues.

(4) A tagged expression stack.

(5) A set of lists of addressing information for dependent tasks.

(6) A set of lists of addressing information for locally declared tasks.

(7) A set of lists of addressing information for locally allocated tasks.

(8) A set of "arms" which contain template indices, continuation addresses, entry indices, and delay intervals, as appropriate, for use by the !select! instruction in implementing the semantics of the Ada select statement.

(9) A set of tagged memories for allocation of space for variables.

(10) A set of tagged memories for allocation of space for formal subprogram parameters.

(11) A set of tagged memories for allocation of space for formal entry parameters.

(12) A connection to each of the effectors (output ports) of the abstract processor on which the virtual processor is running.

The contents of the target machine's two static memories are downloaded from the compiler or copied in toto from a parent to a child during execution of a !createtask! instruction (think of the process as budding). The one containing instructions and string constants is called the codespace. The other is called the templatespace and contains !unittemplate!s. A !unittemplate! will be generated by the compiler for the environment task and for each subprogram, block, accept statement, task, and package found in an Ada program. These will later be referred to as "units". All !unittemplate!s contain certain items of information in common. These are, in order of their appearance within a !unittemplate!:

(1) !stringmax! characters containing the source name of the subprogram, block, task, package, or accept statement whose

occurrence caused the compiler to generate this !unittemplate!. The name is left justified with blank fill. The name of the environment task is "ET". The name of an accept statement is composed of the word "accept " followed by the entry name up to the left parenthesis of the entry index.

(2) The absolute address in codespace of the first instruction (sometimes called an entry point) of the task, subprogram, block, package body's sequence of statements (explicit or implicit), or accept statement. A !return! instruction is generated by the compiler for an accept statement without a "do" part.

(3) The static nesting level of the subprogram, task, block, accept statement, or package. The static nesting level of the environment task is defined to be !minnestingdepth!. All compilations within the Ada program are declarations of units immediately within the environment task; thus, the outermost units of these compilations are defined to have static nesting level !minnestingdepth!+1. The order of these declarations is defined to be a topological sort of the declarations based on the partial order given in the source by the occurrence of "with" clauses. Units nested at greater depths than compilations have greater static nesting levels. The static nesting level of an accept statement or a block is one greater than that of the immediately surrounding subprogram, task, block, accept statement, or package.

(4) A list of absolute addresses of instructions in codespace. Each of these addresses corresponds to an exception declared explicitly or

implicitly. Any particular address is that of the first instruction of the exception handler declared by the subprogram, task, block, accept statement, or package to handle the corresponding exception. If the subprogram, task, block, accept statement, or package declares no handler for an exception, the corresponding address is !nullcodeaddress!. The occurrence of the word "others" in a declaration of exception handlers implies that the addresses corresponding to all exceptions not explicitly listed in that declaration sequence will be of the same instruction.

(5) A keyword of the Pascal type !unittype! specifying for which of a subprogram, task, block, accept statement, or package body this !unittemplate! was generated.

A !unittemplate! generated for a task also contains the value of the task's priority as explicitly given in the Ada source or as assigned by the compiler. !unittemplate!s generated for blocks contain no other information. A !unittemplate! generated for a subprogram or for an accept statement also contains a boolean map for the in (in out) and a boolean map for the out (in out) formal parameters declared explicitly or implicitly (the return value of a function is an implicit parameter) in that subprogram's specification or in the specification of the entry family (an entry declared without an index is a family of one) being accepted. The boolean maps indicate (by true elements) how many and which formal parameter tagged memory !arbnode!s (see below) will be initialized with values from the caller's expression stack and will be pushed back onto that stack upon return. A !unittemplate! generated for a package also contains a value of the Pascal type boolean indicating the veracity of the statement "this package is a library package." The !unittemplate! for the environment task is always at template address !ETtemplate!.

The target machine's codespace is occupied by !instructionunit!s as defined by the Pascal type !instructionunit! below. A very high level description of the semantics in Ada terms of each instruction is also given below. Fields in the type !instructionunit! other than the field opcode represent operands of the appropriate instructions. The unused portion of the instruction memory is initialized to the !arrest! instruction which terminates execution of one or more Ada virtual machines and is not to be generated by the compiler. An !instructionunit! represented by the opcode !dw! is not an instruction but a string constant of 10 characters found in the Ada source in calls to the predefined inline machine code procedure named send_control (see Appendix 1). The target machine's instruction-fetch operates so as to always fetch the instruction having the next greater address unless a branch has taken place using the !destaddr! (or other) operand of an instruction. A return address from a subprogram, accept statement, block, package body's sequence of statements or entry call is referred to as a continuation address and is explicitly loaded before the call. Other continuations may be automatically substituted for these by the target machine's execution. These continuations may be retrieved from the establishment of delays or else parts within select statements, or from exception handler addresses during exception propagation. An instruction address, like a template index, is an integer value which is not less than !nullcodeaddress! (!nulltemplateaddress!). Instruction addresses and template indices can therefore be loaded onto the target machine's expression stack via the !loadintconst! instruction.

Items on the expression stack are described by the Pascal type !arbnode!. Items retain their memory !tag!s while on the expression stack. An item can be placed on the expression stack by a !ref! instruction followed by zero or more !load! instructions, by a !loadcount! or !loadintconst! or !createtask! instruction, or as a result of another instruction which uses other values already on the stack. Many instructions consume values already on the expression stack and leave their results, if any, on the expression stack.

In (in out) actual parameters (explicit or implicit) for entry calls and subprogram calls must be placed on the expression stack prior to the call and results (out or in out parameters or function return values) must be popped off after control has returned normally. Abnormal returns result in parameter space not being occupied on the expression stack.

The target machine has a set of tagged memories (containing !arbnode!s) each of which can be used for storing variables, addresses of variables, the addresses of parameters, the values of task variables, and the values of variables declared as access to task types. Formal parameters of either subprograms or entries do not reside in these memories, but in their own similarly-tagged memories. Within a target machine, these tagged memories are automatically allocated to the corresponding task, and to the subprograms, blocks, accept statements, and sequences of statements of package bodies which it calls, directly or indirectly, in the following proportions: A task has one tagged memory within which the compiler may assign space for the task's local variables, loop indices, temporaries, and task variables. A subprogram has two tagged memories; one for the subprogram's local variables, loop indices, temporaries, and task variables, and the other for the explicit or implicit formal parameters, if any. An accept statement has two tagged memories; one for the accept statement's loop indices, and temporaries, and the other for the formal parameters of the entry family accepted. A block has one tagged memory for its local variables, loop indices, temporaries, and task variables. A package has one tagged memory for the loop indices and temporaries needed by the explicit or implicit sequence of statements of the package body. The variables declared in a package are assigned space by the compiler in the variable tagged memory of the nearest task, subprogram, or block textually surrounding the declaration of the package. The word "textually" here includes the imagined placement of compilations in the discussion of static nesting levels above.

Allocation of space within these tagged memories is assumed to begin at the lowest allowable !offset! and proceed to greater !offset!s. The lowest and highest !offset!s for the variable, etc. tagged memories are !minuserdata! and !maxuserdata!, respectively. The lowest and highest !offset!s for the formal subprogram and entry parameter tagged memories are !lowparmoffset! and !maxparameters!, respectively.

When a new task is created or a new instance of a subprogram, block, accept statement, or package(body's explicit or implicit sequence of statements) is entered, all !arbnode!s in that task's, subprogram's, block's, accept statement's, or package's variable tagged memory have the !tag! !nul!. Thus, initialization of variables is the responsibility of the compiler through instructions generated upon encountering the corresponding "begin".

There are three ways for the compiler to address !arbnode!s in the tagged memories. One is the use of the !index! instruction which adds an integer value to an already existing address. The second method is the use of the !load! instruction where the value loaded is a previously !store!d address. The third method is the use of the !ref! instruction which creates an address from the instruction's operands. The !ref! instruction's !staticnestinglevel! operand specifies the static nesting level of the task, subprogram, block, accept statement, or package one of whose tagged memories contains the location being addressed. The !ref! instruction's !tag! operand specifies which of the tagged memories of the indicated task, subprogram, block, accept statement, or package contains the location being addressed. The only valid values for the !ref! instruction's !tag! operand are !vara! which specifies the tagged memory for variables, loop indices, temporaries, and task variables, !eprm! which specifies the tagged memory for formal entry parameters, and !sprm! which specifies the tagged memory for formal subprogram parameters. The !ref! instruction's !offset! operand supplies the !offset! of the location being addressed within the indicated tagged memory. The !ref!, !load!, and !store! instructions, when used for non-local references, implement the use of shared variables as described with "pragma shared" in the LRM. It is the compiler's duty to make and update local copies of shared variables which are not the targets of that pragma.

The values of task variables and of access variables to task types are identical. These values have !tag! !tska! and are created only by the !createtask! instruction.

The target machine has a set of entry queues which can be referenced by the !entrycallNORMAL!, !entrycallCONDITIONAL!, !entrycallTIMED!, !loadcount! and !setarmaccept! instructions using integer indices from !nullentryindex! to !maxentries!. The association of an entry name in the Ada source with an entry queue index in the target machine is the responsibility of the compiler.

Each task, subprogram, block, accept statement, and package has a list of dependent tasks and a currently accessible dependent task on the list. The !getdependent! instruction changes, to the next dependent on the list, the concept of which dependent is currently accessible. The !resetdependentlist! instruction ensures that the next execution of the !getdependent! instruction will make the first task on the dependent list become the currently accessible dependent. There are two similar lists, one of allocated tasks and one of declared tasks. The concept of currently accessible child task applies at any time to only one of these lists and is moved along that list via the !getownedtask! instruction. To which list the concept of currently accessible child task applies at any time depends on which of the !resetdeclaredtasklist! or the !resetallocatedtasklist! instruction was most recently executed.

A !select! instruction performs most of the actions required by the Ada selective wait. In order to perform this feat, the target machine must be informed of which alternatives of the select statement are open or applicable. The !setarmaccept!, !setarmdelay!, !setarmterminate!, and !setarmelse!, establish for the target machine the presence of open accept alternatives, delay alternative, terminate alternative, and an else part, respectively. The !select! instruction acts upon all of the alternatives (and else parts) established since the execution of the most recent !cleararms! instruction. The upper limit of open accept arms is !maxarms!.

The format of the code and template text file produced by the compiler should be that which would be produced by procedures !write1instruction! and !write1template! (see ~jckNASA/rts/src/proc.i at CSNET address uvacs). All !instructionunit!s appear before the first !unittemplate!, the last !instructionunit! has !opcode! !arrest!, and the last !unittemplate! has !kindofunit! !deadtmplat!. Addresses are implied by the rule that the first instruction in the file is the instruction at !nullcodeaddress!, and the first template in the template file is the template at !nulltemplateaddress! and that addresses increase by 1.

The dependent clean-up code provided in the code sequence examples below must occur beginning at !nullcodeaddress! in order for certain parts of the run-time semantics of Ada to be modeled by the Ada virtual machines.

```
(* DISCLAIMER:                                                  )
(  These constant and type declarations are for organizational  )
(  reference within the context of the text of this manual and  )
(  are not necessarily the operative versions at any given time. )
(  The operative versions are in the Pascal source files        )
(              ~jckNASA/rts/src/const.i                          )
(          and  ~jckNASA/rts/src/type.i   at CSNET address uvacs *)

const

  lowparmoffset      =  1;       maxparameters      =    5;

  stringlow          =  1;       stringmax          =   80;

  minnestingdepth    =  1;       maxnestingdepth    =   20;

  nullcodeaddress    =  0;       codespacesize      = 1000;

  nulltemplateaddress=  0;       numberoftemplates  =   30;

  nullap             =  0;       maxnumberofaps     =   32;

  nullentryindex     =  0;       maxentries         =   10;

  minuserdata        =  0;       maxuserdata        =   20;

  loweffector        =  0;       higheffector       =   49;

  nullexception      =  0;       maxexceptions      =   10;

  taskingerror       =  1;

  ETtemplate         =  nulltemplateaddress;

  maxarms            =  5;

type

  string           = array[stringlow..stringmax]of char;

  nestinglevel     = minnestingdepth..maxnestingdepth;

  codespace        = nullcodeaddress..codespacesize;

  templatespace    = nulltemplateaddress..numberoftemplates;

  apnumber         = nullap..maxnumberofaps;

  exceptionname    = nullexception..maxexceptions;

  effectorrange    = loweffector..higheffector;

  priority         = integer;
```

```
memorytag        = (nul  (* null                                  *)
                   ,intg (* integer                       value   *)
                   ,bool (* boolean                       value   *)
                   ,vara (* variable                      address *)
                   ,eprm (* formal entry      parameter address *)
                   ,sprm (* formal subprogram parameter address *)
                   ,tska (* task address (access value)           *)
                   );
```

```
opcodes              = (cleararms
                        ,arrest
                        ,abortdependent
                        ,aborttask
                        ,ALLOCATE
                        ,delay
                        ,GOTOFROMB
                        ,index
                        ,ref
                        ,return
                        ,RETINBLOC
                        ,awaitactivationdoneforall
                        ,call
                        ,select
                        ,createtask
                        ,dw
                        ,effector
                        ,enableexceptions
                        ,entrycallCONDITIONAL
                        ,entrycallNORMAL
                        ,entrycallTIMED
                        ,getdependent
                        ,getownedtask
                        ,resetdependentlist
                        ,resetdeclaredtasklist
                        ,resetallocatedtasklist
                        ,myactivationisdone
                        ,checkdependent
                        ,letdependentproceed
                        ,iamterminable
                        ,letchildbegin
                        ,removedependent
                        ,activatechild
                        ,setarmaccept
                        ,setarmdelay
                        ,setarmterminate
                        ,setarmelse
                        ,jmp
                        ,brfalse
                        ,brtrue
                        ,raisex
                        ,reraise
                        ,loadcount
                        ,loadintconst
                        ,load
                        ,store
                        ,addinstr
                        ,subinstr
                        ,mulinstr
                        ,divinstr
                        ,modinstr
                        ,eqinstr
                        ,neinstr
,ltinstr
,gtinstr
,leinstr
,geinstr
,andinstr
,orinstr
,xorinstr
,notinstr
);
```

```
instructionunit  = record
                   case opcode : opcodes of
                   dw                  :(s                 : alfa;
                                       );
                   effector            :(area              : effectorrange;
                                         isastring         : boolean;
                                       );
                   ref                 :(staticnestinglevel: nestinglevel;
                                         offset            : integer;
                                         tag               : memorytag;
                                       );
                   entrycallCONDITIONAL,
                   entrycallTIMED      ,
                   setarmaccept        :(tmplate           : templatespace;
                                         continu           : codespace;
                                       );
                   entrycallNORMAL     :(templat           : templatespace;
                                       );
                   getdependent        ,
                   getownedtask        ,
                   setarmdelay         ,
                   setarmelse          :(altpath           : codespace;
                                       );
                   checkdependent      ,
                   abortdependent      ,
                   call                ,
                   letdependentproceed ,
                   letchildbegin       ,
                   removedependent     ,
                   activatechild       :(continue          : codespace;
                                       );
                   setarmterminate     :(chkaddr           : codespace;
                                       );
                   jmp                 ,
                   brfalse             ,
                   brtrue              :(destaddr          : codespace;
                                       );
                   GOTOFROMB           :(numlevelstoexit   : integer;
                                         destaddress       : codespace;
                                       );
                   ALLOCATE            :(size              : integer;
                                       );
                   raisex              ,
                   loadintconst        :(theconst          : integer;
                                       );
                   arrest              ,
                   reraise             ,
                   aborttask           ,
                   index               ,
                   return              ,
                   RETINBLOC           ,
                   awaitactivationdoneforall,
                   select              ,
cleararms                ,
enableexceptions         ,
loadcount                ,
resetdependentlist       ,
resetdeclaredtasklist,
resetallocatedtasklist,
myactivationisdone       ,
iamterminable            ,
delay                    ,
load                     ,
store                    ,
addinstr                 ,
subinstr                 ,
mulinstr                 ,
divinstr                 ,
modinstr                 ,
eqinstr                  ,
neinstr                  ,
ltinstr                  ,
gtinstr                  ,
leinstr                  ,
geinstr                  ,
andinstr                 ,
orinstr                  ,
xorinstr                 ,
notinstr                 :();
createtask     :(template                   : templatespace;
                 masterindex                : nestinglevel;
                 isallocated                : boolean;
                     );
end;
```

```
arbnode             = record
                        case tag : memorytag of
                        nul  :();
                        intg :(i : integer;);
                        bool :(b : boolean;);
                        vara ,
                        eprm ,
                        sprm :(a : variableaddress;); (* compiler cannot generate *)
                        tska :(t : vpname;);          (* values of these types    *)
                        end;

unittype            = (task
                        ,accept
                        ,block
                        ,subprogram
                        ,packagebody
                        );

unittemplate       = record
                     sourcename             : string;
                     firstinstruction       : codespace;
                     staticnestinglevel     : nestinglevel;
                     exceptionhandlerlist : array[exceptionname]of codespace;
                                              (* nullcodeaddress means no handler *)
                     case kindofunit        : unittype of
                     task        :(staticpriority : priority;
                                  );
                     block       :(
                                  );
                     accept      ,
                     subprogram  :(inn,
                                   outt        : packed array
                                                 [lowparmoffset..maxparameters]
                                                 of boolean;
                                  );
                     packagebody :(libraryunit : boolean;
                                  );
                     end;
```

Assuming that the symbol currentinstruction represents the instruction to be interpreted in the form of a value of type !instructionunit!, and that the symbol TOS stands for the !arbnode! on top of the expression stack at any particular point in time, the meaning of the instructions is as follows:

case currentinstruction.opcode of

raisex	Corresponds to the Ada statement "raise x;" where x is the name of some exception and is mapped by the compiler to the integer operand "theconst".
reraise	Corresponds to the Ada statement "raise;".
cleararms	Initiate processing of an accept or selective wait.
setarmaccept	Establishes an open accept alternative for this selective wait. Currentinstruction.continu is the address of the statements following the rendezvous. Currentinstruction.tmplate is the index of the template for the accept statement. Pops from TOS the index of the entry being accepted.
setarmdelay	Establishes an open delay alternative for this selective wait. Currentinstruction.altpath is the address of the code for the delay alternative. Pops from TOS the delay interval.
setarmterminate	Establishes an open terminate alternative for this selective wait. Currentinstruction.chkaddr is the address of the code for the dependent check.
setarmelse	Establishes an else part for this selective wait. Currentinstruction.altpath is the address of the code for the else part.
select	Performs a selective wait using the alternatives established since the most recent !cleararms! instruction and according to Ada's semantics: If there were no open alternatives, raises exception. If there are entries on an open alternative, chooses such an alternative and calls the accept for that alternative. If there are no entries on an open alternative:

	If there are only accept alternatives, waits for an entry call. If there is an else part, branches to the else part. If there is a terminate alternative, branches to the code for the dependent check (if not all dependents are already known to be terminable) or waits for an entry call or task removal. If there is a delay alternative which has expired, branches to that (some) the delay alternative. If there is a delay alternative which has not expired, waits for an entry call or expiration of the delay.
jmp	Branch to currentinstruction.destaddr
brfalse	Pops TOS. If false then branch to currentinstruction.destaddr
brtrue	Pops TOS. If true then branch to currentinstruction.destaddr
ref	Creates an address from currentinstruction.tag (kind of memory containing the object addressed), currentinstruction.staticnestinglevel (of the object addressed), and currentinstruction.offset (within the object's local environment) and pushes it onto TOS.
loadcount	Pops entry index from TOS. Pushes e'count onto TOS.
loadintconst	Pushes an integer value made from currentinstruction.theconst onto TOS.
load	Pops an address from TOS. Pushes the value found at that address onto TOS.
store	Pops a value from TOS. Pops an address from TOS. Stores the value at that address.
entrycallNORMAL	Uses the !inn! parameter map in the template indexed by currentinstruction.templat to pop the actual parameters. That template must be for SOME accept statement which accepts the appropriate entry family; there is no guarantee that that accept statement will be the point of rendezvous in the callee. Pops from TOS the in (in out) actual parameters. The bottommost actual gets the lowest in (in out) formal address. Pops from TOS the entry index. Pops from TOS the called task's address. Places an entry call on that queue.

entrycallCONDITIONAL	Uses the !inn! parameter map in the template indexed by currentinstruction.tmplate to pop the actual parameters. That template must be for SOME accept statement which accepts the appropriate entry family; there is no guarantee that that accept statement will be the point of rendezvous in the callee. Pops from TOS the in (in out) actual parameters. The bottommost actual gets the lowest in (in out) formal address. Pops from TOS the entry index. Pops from TOS the called task's address. Initiates a conditional entry call. If no rendezvous, branches to currentinstruction.continu.
entrycallTIMED	Pops from TOS the delay interval. Uses the !inn! parameter map in the template indexed by currentinstruction.tmplate to pop the actual parameters. That template must be for SOME accept statement which accepts the appropriate entry family; there is no guarantee that that accept statement will be the point of rendezvous in the callee. Pops from TOS the in (in out) actual parameters. The bottommost actual gets the lowest in (in out) formal address. Pops from TOS the entry index. Pops from TOS the called task's address. Initiates a timed entry call. If no rendezvous, branches to currentinstruction.continu.
getdependent	Makes the current dependent the next dependent. If no more dependents, branch to currentinstruction.altpath.
getownedtask	Makes the current child the next child (declared or allocated depending on most recent reset). If no more children, branch to currentinstruction.altpath.
resetdependentlist	Makes the current dependent the first dependent.
resetdeclaredtasklist	Makes the current child the first declared child.
resetallocatedtasklist	Makes the current child the first allocated child.
abortdependent	Aborts the current dependent and branches to currentinstruction.continue.
aborttask	Pops task address from TOS. Aborts that task.

activatechild	If the current child has been created but not yet told to do its activation, tells it to do its activation. Branches to currentinstruction.continue.
awaitactivationdoneforall	Waits until all children have announced they are activated or encountered errors.
letchildbegin	If the current child has announced it is activated, tells it to proceed with its begin statement. Branches to currentinstruction.continue.
myactivationisdone	Announces to parent (not master) that this child is activated. Awaits permission to proceed with begin statement.
checkdependent	Checks whether the current dependent is ready to terminate. If so, holds the dependent in limbo and branches to currentinstruction.continue.
letdependentproceed	Takes the current dependent out of limbo due to a !checkdependent! instruction. Branches to currentinstruction.continue.
iamterminable	Sets an internal flag announcing the task may terminate.
removedependent	Eradicates the current dependent. Branches to currentinstruction.continue.
delay	Pops delay interval from TOS. Delays at least that long.
ALLOCATE	NOT IMPLEMENTED YET (used for dynamic allocation of other than tasks).
GOTOFROMB	NOT IMPLEMENTED YET (used for a goto within a block whose destination is outside of that block).
RETINBLOC	NOT IMPLEMENTED YET (used for a return statement which is textually within a block).
return	Propagates any pending/unhandled exceptions to the calling unit and/or surrounding unit or nowhere as appropriate for Ada's semantics. If returning from a task, becomes terminated and awaits removal by its master. If returning from an accept, either raises tasking_error in the caller of the entry or copies the out (in out) formal entry

	parameters back onto the TOS of the caller of the entry leaving lowest formal addressed bottommost. If returning from an accept, allows the caller of the entry to proceed. If returning from a subprogram, pushes the out (in out) formal parameters onto TOS leaving lowest formal addressed bottommost. If not returning from a task, resumes at the continuation address saved at the call or select.
call	Pops from TOS index of the template for the unit being called. (The index is not a fixed operand in order that generic subprogram parameters may be implemented.) Uses as continuation address (return address) currentinstruction.continue If the unit is a subprogram (determined from the template), pops from TOS the actual parameters into the in (in out) formal parameters. The bottommost actual gets the lowest formal address.
createtask	Pops from TOS the integer number of the abstract processor on which the instruction creates a task with the unit at nesting level currentinstruction.masterindex as master and using the template indexed by currentinstruction.template. Currentinstruction.isallocated determines whether the new task becomes a declared or an allocated child. The task's address is left on TOS.
enableexceptions	On entry to a unit, exceptions are inhibited. This allows subsequent exceptions to be raised. Any exceptions which would otherwise have been raised previously in this unit are raised here.
effector	Pops from TOS an intg or bool arbnode. If the abrnode is bool or not currentinstruction.isastring, uses the arbnode's value, otherwise considers the arbnode's value to be a constant string address. Writes the value or the first 10 characters of the string onto the AP's port addressed by currentinstruction.area
index	Pops an index from TOS. Pops an address from TOS. Pushes onto TOS a new address formed from the first increased by the index.

addinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes their sum onto TOS.
subinstr	Pops an integer minuend from TOS. Pops an integer subtrahend from TOS. Pushes the difference onto TOS.
mulinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes their product onto TOS.
divinstr	Pops an integer divisor from TOS. Pops an integer dividend from TOS. Pushes the integer quotient onto TOS.
modinstr	Pops an integer divisor from TOS. Pops an integer dividend from TOS. Pushes the integer remainder onto TOS.
eqinstr	Pops a value from TOS. Pops a value from TOS. If their tags are different, pushes a boolean false onto TOS, else if they are integer or boolean, pushes the boolean result of comparing them for equality onto TOS, else raises exception.
neinstr	Pops a value from TOS. Pops a value from TOS. If their tags are different, pushes a boolean true onto TOS, else if they are integer or boolean, pushes the boolean result of comparing them for inequality onto TOS, else raises exception.
ltinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes onto TOS the boolean result of comparing them for (the second less than the first).
gtinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes onto TOS the boolean result of comparing them for (the second greater than the first).
leinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes onto TOS the boolean result of comparing them for (the second less than or equal to the first).
geinstr	Pops an integer from TOS. Pops an integer from TOS. Pushes onto TOS the boolean result of comparing them for (the second greater than or equal to the first).

andinstr	Pops a boolean from TOS. Pops a boolean from TOS. Pushes onto TOS the boolean result of comparing them for (both true).
orinstr	Pops a boolean from TOS. Pops a boolean from TOS. Pushes onto TOS the boolean result of comparing them for (at least one true).
xorinstr	Pops a boolean from TOS. Pops a boolean from TOS. Pushes onto TOS the boolean result of comparing them for (exactly one true).
notinstr	Pops a boolean from TOS. Pushes its logical complement onto TOS.

end of case

Sample uses of the instructions:

```
/---------------------------------------------------------------------------
| BY USING THESE CODE SEQUENCES, THE COMPILER MAKES THE VP LOOK LIKE AN Ada |
| MACHINE. NOTE THAT THE SEQUENCES ARE GIVEN IN A SYMBOLIC PSEUDO-ASSEMBLY  |
| CODE FORM WHICH THE COMPILER IS *NOT* TO SEND TO THE INTERPRETER.         |
| A legend follows on this page.                                            |
---------------------------------------------------------------------------/
+----------------------+
|  Ada source example  |
+----------------------+
      Assembly language translation where           Pertinent information
      digits: is a label                            in template space i.e.
      {}      surrounds an indication that          symbolic templates
              other code should be expanded
              at that location
      <>      surrounds a reference to a label
              in code space of template space.
              this is an "address of" operator
============================================================================
+---------------------+
| accept e1(i)(p);    |
| nxtstmt;            |
+---------------------+
    cleararms                                        ts1: [
    {eval e1(i)}                                          |"accept e1(i)"
    setarmaccept        (<ts1>,<2>)                       |<1>
    select                                                |1+(nesting level of unit
 1:                                                       |   containing this accept)
    return                                                |(exceptionhandlerlist
 2:                                                       | contains all zeros)
    jmp                <3>                                |ACCEPT
 3:                                                       |(parameter maps depend on (p))
    {nxtstmt}                                             ]
===========================================================================
```

```
+-----------------------+
| accept e1(i)(p) do    |
|         stmt1;        |
| end e1;               |
| nxtstmt;              |
+-----------------------+
    cleararms                                        ts1: [
    {eval e1(i)}                                          |"accept e1(i)"
    setarmaccept       (<ts1>,<2>)                        |<1>
    select                                                |1+(nesting level of unit
 1: {stmt1}                                               |   containing this accept)
    return                                                |(exceptionhandlerlist
 2:                                                       | depends on text in stmt1)
    jmp                <3>                                |ACCEPT
 3:                                                       |(parameter maps depend on (p))
    {nxtstmt}                                             ]
===========================================================================
+----------------------------------------------------------+
|                                                          |
|   select                                                 |
|           when b1 =>        accept e1(i)(p) do           |
|                                     stmt1;               |
|                             end e1;                      |
|                             stmt2;                       |
|           or                                             |
|           when b2 =>        accept e2(i)(p) do           |
|                                     stmt3;               |
|                             end e2;                      |
|                             stmt4;                       |
|           or                                             |
|           when b3 =>        delay x1;                    |
|                             stmt5;                       |
|           or                                             |
|           when b4 =>        delay x2;                    |
|                             stmt6;                       |
|           or                                             |
|           when b5 =>        terminate;                   |
|           else                                           |
|                             stmt7;                       |
|   end select;                                            |
|   nxtstmt;                                               |
|                                                          |
+----------------------------------------------------------+
```

```
   cleararms
   {eval b1}
   brfalse            <1>
   {eval e1(i)}
   setarmaccept       (<ts1>,<13>)
1:
   {eval b2}
   brfalse            <2>
   {eval e1(i)}
   setarmaccept       (<ts3>,<15>)
2:
   {eval b3}
   brfalse            <3>
   {eval x1}
   setarmdelay        <16>
3:
   {eval b4}
   brfalse            <4>
   {eval x1}
   setarmdelay        <17>
4:
   {eval b5}
   brfalse            <5>
   setarmterminate    <7>
5:
   setarmelse         <18>
6:
```

```
ts1: [
     |"accept e1(i)"
     |<12>
     |1+(nesting level of unit
     |   containing this accept)
     |(exceptionhandlerlist
     | depends on text in stmt1)
     |ACCEPT
     |(parameter maps depend on (p))
     ]
ts3  [
     |"accept e1(i)"
     |<14>
     |1+(nesting level of unit
     |   containing this accept)
     |(exceptionhandlerlist
     | depends on text in stmt2)
     |ACCEPT
     |(parameter maps depend on (p))
     ]
```

```
    select
 7: resetdependentlist
 8: getdependent       <11>
    checkdependent     <8>
 9: resetdependentlist
10: getdependent       <6>
    letdependentproceed <10>
11: iamterminable
    jmp                <9>

12: {stmt1}
    return
13: {stmt2}
    jmp                <19>
14: {stmt3}
    return
15: {stmt4}
    jmp                <19>
16: {stmt5}
    jmp                <19>
17: {stmt6}
    jmp                <19>
18: {stmt7}
    jmp                <19>
19: {nxtstmt}
==========================================================================
+----------+
| delay x; |
| nxtstmt; |
+----------+
    {eval x}
    delay
    {nxtstmt}
==============================================================================
```

```
+-------------+
| t.e(i)(p);  |
| nxtstmt;    |
+-------------+
    {eval t}
    {eval e(i)}
    {eval p}
    entrycallNORMAL    <some template accepting t.e()>
    {pop resulting parameters p' from TOS}
    {nxtstmt}
==============================================================================
```

```
+------------------+
| select           |
|      t.e(i)(p);  |
|      stmt1;      |
| else             |
|      stmt2;      |
| end select;      |
| nxtstmt;         |
+------------------+
    {eval t}
    {eval e(i)}
    {eval p}
    entrycallCONDITIONAL    (<some template accepting t.e()>,<1>)
    {pop resulting parameters p' from TOS}
    {stmt1}
    jmp                <2>
 1: {stmt2}
 2: {nxtstmt}
==============================================================================
+-----------------+
| select          |
|      t.e(i)(p); |
|      stmt1;     |
| or              |
| delay x;        |
|      stmt2;     |
| end select;     |
| nxtstmt;        |
+-----------------+
    {eval t}
    {eval e(i)}
    {eval p}
    {eval x}
    entrycallTIMED     (<some template accepting t.e()>,<1>)
    {pop resulting parameters p' from TOS}
    {stmt1}
    jmp                <2>
 1: {stmt2}
 2: {nxtstmt}
==============================================================================
```

```
+-----------------------------------------------------------------------------+
| {block, subprogram, or sequence_of_statements_of_a_package_body call};      |
| nxtstmt;                                                                    |
+-----------------------------------------------------------------------------+
    {eval p} -- if a subprogram
    loadintconst             (template index)
    call                     <1>
    {code for the block goes here, if calling a block}
 1:
    {pop resulting parameters p' from TOS} -- if a subprogram
    {nxtstmt}
==============================================================================
+-------------------+
|  (unit prologue)  |
+-------------------+
 1: -- entry point
    -- The next 3 instructions are repeated for each task declared)
    ref                {task variable}
    loadintconst       {AP_number}
    createtask         (template_index,master's_nesting_level,false)
    store
    -- The next 3 instructions are repeated for each task allocated)
    ref                {access variable}
    loadintconst       {AP_number}
    createtask         (template_index,master's_nesting_level,true)
    store
    {whatever else counts as activation}
    -- the following instruction occurs only in task bodies
    myactivationisdone
    -- here up to the enableexceptions instruction is optional
    -- if the declarations declare/allocate no tasks
    -- NOTE: a compiler is permitted to use more intelligence here
    --       omitting sections which don't apply if tasks are only
    --       declared or only allocated or neither in the unit.
    resetdeclaredtasklist
 2: getownedtask       <3>
    activatechild      <2>
 3: resetallocatedtasklist
 4: getownedtask       <5>
    activatechild      <4>
 5: awaitactivationdoneforall
    resetdeclaredtasklist
 6: getownedtask       <7>
    letchildbegin      <6>
 7: resetallocatedtasklist
 8: getownedtask       <9>
    letchildbegin      <8>
 9: enableexceptions
    {set up initial values of variables}
===========================================================================
+-----------------------+
|  t := new tasktype;   |
|  nxtstmt;             |
+-----------------------+
    ref                 (t)
    loadintconst        {AP_number}
    createtask          (template_index_for_tasktype,master's_nesting_level,true)
    store
    resetallocatedtasklist
 1: getownedtask        <2>
    activatechild       <1>
 2: awaitactivationdoneforall
    resetallocatedtasklist
 3: getownedtask        <4>
    letchildbegin       <3>
 4: {nxtstmt}
============================================================================
```

```
+---------------------------------------------------+
|  {dependent clean-up code at !nullcodeaddress!}   |
+---------------------------------------------------+
    -- all units (except accepts) jump here rather than return
 1: resetdependentlist
 2: getdependent               <4>
    checkdependent             <2>
    resetdependentlist
 3: getdependent               <1>
    letdependentproceed        <3>
 4: resetdependentlist
 5: getdependent               <6>
    removedependent            <5>
 6: return
============================================================================
-- hand coded example program
 task body ET is

    procedure main is

       task a is
       end a;

       task b is
          entry e;
       end b;

       task body b is
          begin -- b
             loop
                accept e do
                 send_control(2,"B");
                end e;
             end loop;
          end b;

       task body a is
          begin -- a
             loop
                b.e;
                send_control(1,"A");
             end loop;
          end a;

       begin -- main
          null;
       end main;

 begin -- ET
    main;
 end ET;
```

```
-- /------------
-- | HAND-CODED |
-- ------------/
   -- almost all bodies are going to jmp here
 5: resetdependentlist
 6: getdependent              <8>
    checkdependent            <6>
    resetdependentlist
 7: getdependent              <5>
    letdependentproceed       <7>
 8: resetdependentlist
 9: getdependent              <10>
    removedependent           <9>
10: return
 0: enableexceptions
    loadintconst              <ts17>
    call                      <5>
    ------------------------------------
11: myactivationisdone
    enableexceptions
12: cleararms
    loadintconst              1
    setarmaccept              (<ts13>,<14>)
    select
    ------------------------------
13: loadintconst              <23>
    effector                  (2,true)
    return
    ------------------------------
14: jmp                       <12>
    jmp                       <5>
    ------------------------------------
```

```
ts_0: [
      |"ET"
      |<0>
      |1
      |{lots of zeros}
      |TASK
      |1
      ]
ts17: [
      |"main"
      |<17>
      |2
      |{lots of zeros}
      |SUBPROGRAM
      |{maps all false}
      ]
ts15: [
      |"a"
      |<15>
      |3
      |{lots of zeros}
      |TASK
      |1
      ]
ts11: [
      |"b"
      |<11>
      |3
      |{lots of zeros}
      |TASK
      |1
      ]
```

```
15: myactivationisdone                                     ts13: [
    enableexceptions                                             |"accept e"
16:                                                              |<13>
    ref                   (2,3,vara) -- {task variable b}        |4
    load                                                         |{lots of zeros}
    loadintconst          1                                      |ACCEPT
    entrycallNORMAL       <ts13>                                 |{maps all false}
    loadintconst          <22>                                   ]
    effector              (1,true)
    jmp                   <16>
    jmp                   <5>
    ----------------------------------------------
17: ref                   (2,2,vara) -- {task variable a}
    loadintconst          1
    createtask            (2,<ts11>,false)
    store
    ref                   (2,3,vara) -- {task variable b}
    loadintconst          2
    createtask            (2,<ts13>,false)
    store
    resetdeclaredtasklist
18: getownedtask          <19>
    activatechild         <18>
19: awaitactivationdoneforall
    resetdeclaredtasklist
20: getownedtask          <21>
    letchildbegin         <20>
21: enableexceptions
    jmp                   <5>
    ----------------------------------------------
22: dw                    "A          "
23: dw                    "B          "
    ----------------------------------------------
==============================================================================
+-----------------------------------------+
| -- hand coded example program           |
|  task body ET is                        |
|                                         |
|     procedure main is                   |
|                                         |
|        begin -- main                    |
|           send_control(1,"S");          |
|        end main;                        |
|                                         |
|  begin -- ET                            |
|     main;                               |
|  end ET;                                |
+-----------------------------------------+
 -- /------------                                          ts_0: [
 -- | HAND-CODED |                                               |"ET"
 -- ------------/                                                |<0>
    -- almost all bodies are going to jmp here                   |1
 5: resetdependentlist                                           |{lots of zeros}
 6: getdependent            <8>                                  |TASK
    checkdependent          <6>                                  |1
    resetdependentlist                                           ]
 7: getdependent            <5>                            ts11: [
    letdependentproceed     <7>                                  |"main"
 8: resetdependentlist                                           |<11>
 9: getdependent            <10>                                 |2
    removedependent         <9>                                  |{lots of zeros}
10: return                                                       |SUBPROGRAM
 0: enableexceptions                                             |{maps all false}
    loadintconst            <ts11>                               ]
    call                    <5>
    ----------------------------------------
11: enableexceptions
    loadintconst            <12>
    effector                (1,true)
    return
    -- it is not required that a subprogram do
    --    jmp               <5>
    -- if it *cannot* have dependents.
    ----------------------------------------
12: dw                      "S         "
    ----------------------------------------
=============================================================================
```

APPENDIX 2

FAULT TOLERANT DISTRIBUTED SYSTEMS USING Ada

John C. Knight John I. A. Urquhart
Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville
Virginia, 22901

Abstract

This paper discusses the use of Ada on distributed systems in which failure of processors has to be tolerated. We assume that communication between tasks on separate processors will take place using the facilities of the Ada language, primarily the rendezvous. We show that there are numerous aspects of the language which make its use on a distributed system very difficult. The issues are raised from the desire to be able to recover, reconfigure, and provide continued service in the presence of hardware failure. For example, if a rendezvous takes place between two tasks on different processors, failure of the processor executing the serving task will cause the calling task to be permanently suspended because the rendezvous will never end. Extensive modifications to the execution support required for Ada are proposed which provide all the necessary facilities for programs written in Ada to withstand arbitrary processor failure. Mechanisms are suggested to allow processor failure to be detected and for tasks which would be permanently suspended to be released. Provided the required program structures are used, continued processing can be provided.

* Ada is a trademark of the U.S. Department of Defense. This work was supported by NASA grant number NAG-1-260.

Introduction

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

In this paper we discuss the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processor they are executing on. We assume that communication between tasks on separate processors will take place using the facilities of the Ada language, primarily the rendezvous. It would be possible to build a separate set of facilities for communication between processors and treat the software on each machine as a separate program. This is pointless however since such facilities would necessarily duplicate the existing facilities of the rendezvous.

We also assume that processors in the system may fail at arbitrary times and that the system is required to continue providing service following such failures. It is shown that this causes considerable difficulties for Ada programs. Solutions to these problems are suggested.

The Need To Cope With Hardware Failure

The kind of architecture we expect to be in common use for embedded systems in the future in shown in figure 1. It is based on the use of a high-performance data bus which links several processors. Each processor is equipped with its own memory. Devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors. Thus these devices would be accessible from each processor.

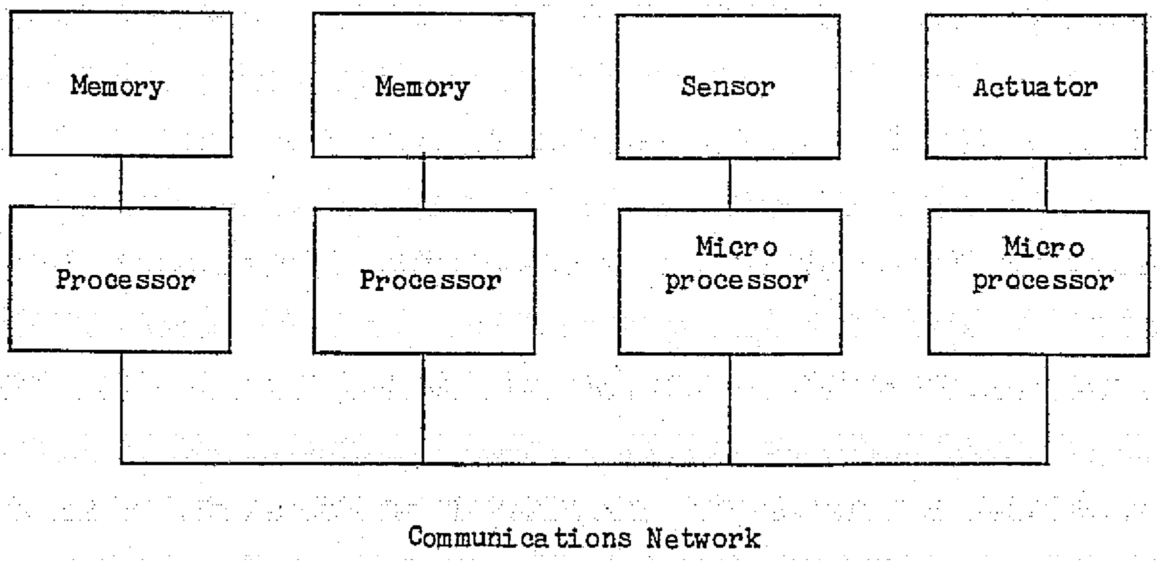


Figure 1 - Distributed Architecture

A great deal of research has been undertaken in recent years to produce computer architectures with high reliability such as the SIFT[1] and FTMP[2] machines. Why then should there be any concern for software structures which are able to cope with hardware failure? There are several reasons:

(1) The architectures of highly-reliable systems are very complex. Such machines are, in effect, highly-parallel multiprocessors and their reliability is achieved by parallelism. These architectures are the subject of current experimentation and are still unproven.

(2) Even though designed for reliability, these machines may still fail.

(3) Physical damage could cause a processor to fail no matter how carefully the processor was built. Fire, structural failure, excess or unexpected vibration, and so on, could cause enough damage that even a highly-parallel machine would be unable to continue.

(4) Electrical damage from unexpected lightning effects could cause a processor to fail.

(5) In a situation where a major power failure occurred, reserve power might only be provided for some subset of the processors. The switch from full power to limited reserve power might be orderly in which case very sophisticated reconfiguration might be possible. However, it might be preferable to use a single, consistent mechanism for recovery to cope with all cases.

(6) Unmanned spacecraft frequently make extensive use of computers but are usually unable to pay the weight and power costs of extensive redundancy (such as in quad redundancy). Reconfigurable distributed systems designed to cope with processor failure is an attractive alternative. If the design includes higher processing power than is absolutely needed, and tasks exist which are not essential to mission success, then some loss of hardware followed by reconfiguration may allow the mission to continue successfully.

Thus, although great care may be taken with the construction of a digital computer system, failure may still occur. At least with a distributed system there is the possibility that if part of the system is lost, what remains could continue to provide service.

Initially, we assume that communication between processors on a distributed system will be implemented using layers of software that conform for the most part to the ISO standard seven-layer Reference Model[3]. The hardware topology that is used for a distributed system need have very little impact on the programming of the system at the application-layer level. In principle, provided the implementation knows how tasks are distributed to processors and how communication is to be achieved, the various tasks can synchronize and communicate at will with no knowledge of their location.

The kinds of hardware failure that we are concerned with are not addressed by the ISO protocol. The ISO protocol is concerned with communications failures such as dropped bits caused by noise, loss of messages or parts of messages, etc. Also, situations such as a processor "slowing down" or incorrectly computing results are not of interest here (though they are important nevertheless). We assume that such events are taken care of by hardware checking within the processor. The only class of faults not dealt with elsewhere is the total loss of a processor or bus with no warning. These are the difficulties we will attempt to deal with.

Ada Issues And Difficulties

In this section, some of the difficulties with the use of Ada on a distributed system are described. We examine only the simple rendezvous and the timed entry call. Lack of space precludes examination of the entire language but this as been done elsewhere^{4}. Proposed solutions to some of the problems raised here are given in a later section.

Simple Rendezvous

A simple rendezvous in Ada consists of a calling task C making an entry call, S.E, to a serving task S, which contains an accept statement for the entry E. The syntax is shown in figure 2. The semantics of the language require that if the call is made by C before the accept is reached by S, C is suspended until the accept is reached. If S reaches the accept before the call is made by C, S is suspended until the call is made. In either case, C remains suspended until the rendezvous itself is complete.

In order to look at the issues arising from a rendezvous in which the tasks involved are on different processors, it is necessary to

```
        Calling Task C          Serving Task S

              .                   ACCEPT E DO
              .                       .
              S.E;                    .
              .                       .
              .                   END E;
```

Figure 2 - Syntax Of A Simple Rendezvous.

specify an implementation of the rendezvous at the message passing level. Only the simple case of a task C calling an entry E in a serving task S will be considered. Further, it will be assumed that the call is made before S has reached the corresponding accept; the case where the server waits at its accept is similar. One possible message sequence is shown in figure 3. The numbers inside brackets represent points of interest in the message sequence.

```
      Caller C         Messages          Server S

        S.E;                               [4]
               PUT_ONTO_QUEUE------->
        [1]                                [5]
                                         ACCEPT E DO
               <---------CHECK_CALLER
        [2]                                [6]
               CHECK_CALLER_REPLY-->
        [3]                                [7]

                                         END E;
               <-RENDEZVOUS_COMPLETE
```

Figure 3 - Rendezvous Message Sequence

The calling task C asks to be put onto the queue for entry E. When S reaches its accept for E, it sees that C is on the queue. At this point S checks to see if C has been aborted. When the CHECK_CALLER message arrives at C, C can be considered to be engaged in the rendezvous. When the reply reaches S, S will start to execute the rendezvous code. When it is completed the RENDEZVOUS_COMPLETED message would awaken C which would continue.

Effects Of Processor Failure On Rendezvous

Using this implementation of a simple rendezvous, what happens if either processor fails? There are seven cases of interest and they are discussed below. The numbers refer to figure 3.

CALLER FAILS AT — EFFECT ON SERVER

[1] The message CHECK_CALLER will not be able to arrive. The effect on the sender should be equivalent to a negative reply to the CHECK_CALLER message (e.g. if the caller had been aborted, but not yet removed from the queue). That is the server would remove the caller from the queue and remain waiting at the accept.

[2] The message CHECK_CALLER arrives, then the caller's processor fails and the reply is never sent. If the server cannot find out that there has been a failure, the server will be trapped waiting for the message CHECK_CALLER_REPLY.

[3] When the caller's processor fails during the rendezvous, the situation is similar to the case where the caller is aborted during the rendezvous. In both cases the server can continue. At the end of the rendezvous the RENDEZVOUS_COMPLETE message cannot arrive; as before, if the server can detect that there has been a failure, the server can continue.

SERVER FAILS AT	EFFECT ON CALLER
[4]	The message PUT_ONTO_QUEUE cannot arrive. The situation is similar to the case where the server is abnormal.
[5]	Here the caller is on an entry queue when the server's processor fails. As before if the failure cannot be detected the caller will be trapped.
[6]	The message CHECK_CALLER has arrived at the caller who now considers that the rendezvous has started; the reply cannot arrive. Again without failure detection the caller will be trapped. (Note that even if the caller were using a timed entry call, the timer would have been turned off by the message CHECK_CALLER.)
[7]	The server's processor fails during the rendezvous. (Timed and conditional entry calls give no protection as they time the delay to the start of the rendezvous.) Again the caller is trapped unless the failure can be detected.

The serving task is not seriously affected when the calling task's processor fails. At worst, time is lost doing work for a task that is not there to receive it. The calling task is in a much worse situation when the server's processor fails. If the rendezvous has not already started the caller will wait on the entry queue for ever. In principle, timed entry calls (discussed below) can handle this situation, if they are implemented by having the calling task do the timing. If the serving task's processor fails after the rendezvous has started, even a caller who has made a timed or conditional entry call will be trapped for ever.

What the caller would like to have, and what even timed and conditional entry calls do not give, is a guarantee that after a certain time it will be possible to proceed. The rules of the language imply that once a rendezvous has started the caller cannot withdraw until it is completed. Clearly withdrawal is necessary when the server's processor fails.

Rendezvous By Timed Entry Call

Timed entry calls are intended to solve some of the problems raised above. In fact, they raise further problems about their meaning and their implementation.

The semantics of the timed entry call appear to be quite straightforward:

> A timed entry call issues an entry call that is canceled if a rendezvous is not started within a given delay[5].

In a distributed system, however, messages will take time to get from a task on one processor to a task on another. Even if the underlying message passing system can guarantee that a message will eventually arrive correctly, this will be implemented at a lower level by a protocol which may involve acknowledgement of messages, and the resending of messages that have been lost. A message can certainly be delayed for some arbitrary length of time. Even physical separation of the processors may impose a significant delay.

One possible interpretation of the timed entry call would be to count the total time until the rendezvous is started. Message passing time and time on the entry queue would be included. This interpretation probably has to be ruled out because the language definition states that a timed entry call with a delay of zero is the same as a conditional entry call. If the delay included both message passing time and time on the queue, a delay of zero would be impossible and a timed entry call with a delay of zero would never succeed.

Another interpretation of the delay in a timed entry call is that it is just the delay on the entry queue. This has a meaning when the specified delay is zero but the important implementation question becomes "who is to do the timing". The calling task cannot do the timing. It is impossible for it to measure waiting time on the entry queue accurately since the message passing time can vary. Thus it is essential that the serving task does the timing.

A timed entry call gives protection against having to wait too long on the entry queue. However, what the task issuing the call needs is some guarantee that it will not be trapped in an attempt to communicate, and forced to miss a deadline. In principle, it does not matter to the task whether the time is spent waiting on a queue or sending messages.

If the timed entry call is implemented by having the server do the timing and the server's processor fails before a rendezvous is started the caller will be trapped. Even if there is no failure the calling task must wait for a message from the server. If the server is doing the timing, that message may need to be re-sent several times, so the calling task may have to wait an arbitrary time.

If the calling task were able to do the timing then an infinite wait could be avoided when the server's processor failed. As we have noted however, this method of timing is unrealistic when queue time is being measured.

For a distributed system, we conclude that there are many problems with the timed entry call. It does not provide the kind of protection that is desirable, the semantics are unclear, and it is very difficult to implement. An analysis of the message traffic necessary for the timed entry call can be performed that is similar to that shown in figure 2. The issues which arise when considering failure are similar but more extensive than those arising in the simple rendezvous.

Other Issues

When the possibility of processor failure is considered, many other aspects of Ada present difficulties similar to those outlined above. For example, problems arise with conditional entry calls, accessing global variables, task elaboration, and task termination. The problems are of a similar cause; namely the prospect that no reply will ever be received to a message sent because of the failure of the processor expected to generate the reply.

Other areas which cause difficulty are global variables and the task master/dependent relationship. If data which is global to more than one task is ever used to share data, loss of the processor containing the data causes great difficulty. Finally, when the processor executing a master task is lost, the dependents of the master should be aborted since loss of the processor executing the master is equivalent to the master being aborted. This raises special difficulties when the master is the main program.

Failure Detection And Signaling

Processor failure cannot be dealt with unless it can be detected. Details of the failure must also be supplied to the software which is to cope with the reconfiguration. How can Ada programs detect hardware failure and what information is needed for reconfiguration? In this section, we present an approach to hardware failure detection and the rational for its choice.

Failure Detection

Failure detection could be performed by hardware facilities over and above those provided for normal system operation. Alternatively, failure could be detected by system software. The hardware option is less desirable because it requires additions to existing or planned systems and the detection hardware itself could fail. We suggest therefore the use of software failure detection.

Software failure detection can be either passive or active. A passive system might rely on tasks assuming that failure had occurred if actions did not take place within a "reasonable" period of time i.e. timing out. Alternatively, a passive system could require that all messages passed between tasks on separate processors be routinely acknowledged. Thus the sender would be sure that the receiver had the message and presumably would act on it. This is a particularly simple case of timing out since failure has to be assumed if no acknowledgement is received.

The disadvantages of passive detection are:

(1) Timing out assumes an agreed-upon upper limit for response time.

(2) A failed processor will not be detected until communication is attempted and this may be long after the failure has occurred.

Upper bounds on response time may be hard to determine. Very complex situations can arise from an incorrect choice. The reason for a lack of response from a task on another processor may not be failure of that processor but merely a temporary rise in its workload. The consequences could be an assumption by one processor that another had failed, followed by reconfiguration to cope with the loss. Clearly, if this assumption is wrong, two processors could begin trying to provide the _same_ service.

Being unaware that a processor had failed will lead to a loss of the service it was providing until the failure is noticed. In a system with many processors each providing relatively few services, the amount of inter-processor communication might be quite low. Thus, a failed processor may go unnoticed for so long a time that damage to the equipment being controlled might result from its lack of service.

It is for these reasons that we reject passive software failure detection and suggest the use of active software failure detection. In an active system, some kind of inter-processor activity is required "periodically" and if it ceases, failure is assumed. The messages which are passed are usually referred to as <u>heartbeats</u>.

As soon as a heartbeat disappears, the remaining processors in the system will be aware that a failure has occurred and they will know which processor has failed. This information must be transmitted to the software running on each remaining processor so that reconfiguration can take place. The information is available to the run-time support software in some internal format, but how should it be transmitted to the Ada software?

One approach is to use the language's exception mechanism, and for the run-time system to generate an exception on each processor. Another approach is to view the required signal as being very like an interrupt, and transmit the information to the Ada software in the way that interrupts are transmitted; namely by an entry call. We prefer this latter approach because it can take place in parallel with any activity that might already be going on. A task designed to cope with reconfiguration could be present on each processor and suspended at an accept statement for the entry which will be called when a failure occurs. This allows each processor to have a "focal point" for reconfiguration. If exceptions are used, the correct placement of the necessary handler is difficult to determine because it will be impossible to know what tasks will be engaged in what activities when the exception is generated.

Thus we propose that a special task be defined on each processor which will contain entries for each hardware component whose failure requires some processing. This task will be normally suspended on the accept statements for the special entries. When a failure occurs, an entry call will be generated and the task will then be activated. It will contain code following each accept statement to handle reconfiguration.

It is not sufficient to detect failure and inform the software of the failure using the methods described above. As discussed above, many Ada language elements (particularly the rendezvous) can lead to situations in which one task is permanently suspended if the processor on which another task is executing fails. These tasks which would be permanently suspended must somehow be released.

The mechanism which we propose to cope with this situation is shown in figure 4. Whenever any communication takes place between tasks on different processors, the run-time support systems on the processors involved record the details of the communication in message logs. Whenever a failure is detected, each processor checks its message log to see if any of its tasks would be permanently suspended by the failure. If any are found, they are sent "fake" messages. They are called fake because they are constructed to appear to come from the failed processor but clearly do not. The message content is usually equivalent to that which would be received if the task on the failed processor had been aborted. Thus for example, if a simple rendezvous is taking place and the processor executing the server fails, the exception TASKING_ERROR is raised in the caller. In this way, each processor is able to ensure that none of its tasks is permanently suspended. However, it is the responsibility of the tasks themselves to ensure that their subsequent

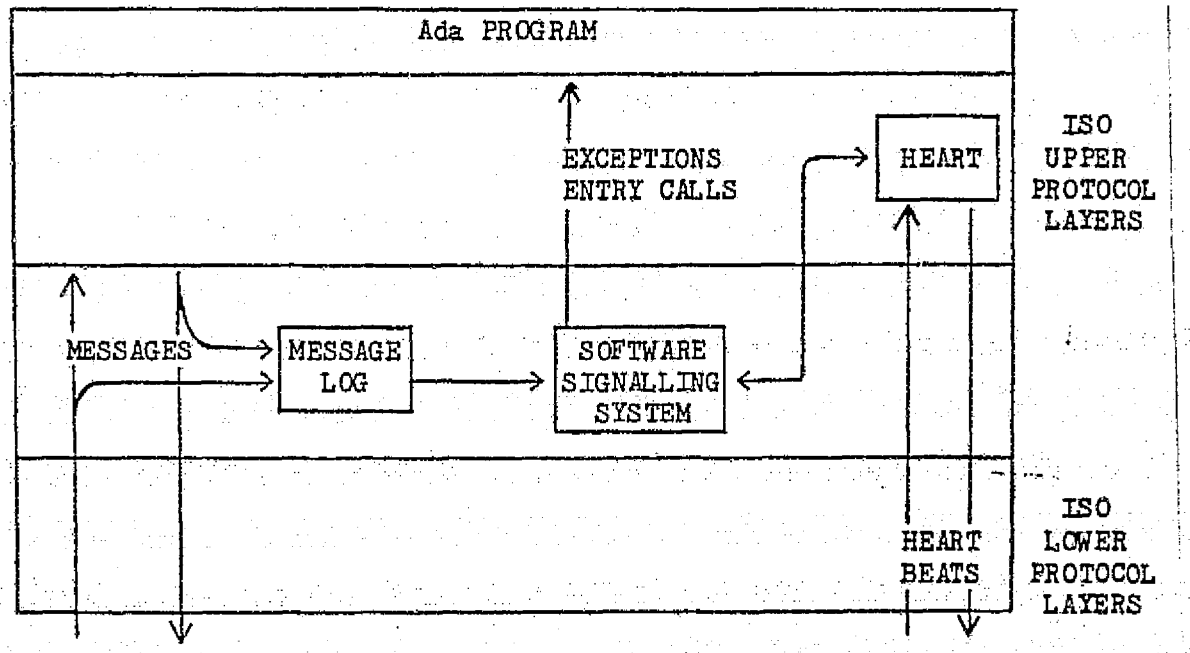

Ada PROGRAM
EXCEPTIONS
ENTRY CALLS
HEART
ISO UPPER PROTOCOL LAYERS
MESSAGES
MESSAGE LOG
SOFTWARE SIGNALLING SYSTEM
HEART BEATS
ISO LOWER PROTOCOL LAYERS

Figure 4 - Implementation

actions are appropriate.

Fault Tolerant Programming Example

This is a very simple example designed to illustrate some of the ideas discussed above. In a typical Ada application, the program would be much larger and would have to take into account all the language features mentioned.

The example consists of a calling task CALLER which operates on one processor (CPU 1) and a serving task SERVER which operates on another processor (CPU 2). The calling task does some real-time processing and calls an entry in the serving task in order to get some kind of service. The program is written to cope with failure of either processor. Alternates are provided for the calling and the serving tasks and a reconfiguration task is present on each processor.

Normally only the calling and serving tasks are executing and a fault-intolerant version of this example would consist of just these two tasks. If processor one fails then it is necessary to start an alternate calling task on processor two. Similarly, if processor two fails it is necessary to start an alternate serving task on processor one.

The alternates are present on the required machines when the program starts execution. Each alternate is waiting on an entry named ABNORMAL_START so that they do no processing while both processors are operational. When one processor fails, the run-time system generates an entry call on the other processor to an entry in its task RECONFIGURE_CPU_i (where i is the processor number). This task then calls the ABNORMAL_START entry for the alternate which is needed and processing is able to continue. Entries are defined in RECONFIGURE_CPU_i for each component that might fail. In this example, each machine is only interested in the failure of the other so only one entry is defined in each reconfiguration task.

If a rendezvous is in progress when the failure occurs, then the serving task need not care that the calling task has been lost, and the rendezvous can complete. The calling task will care if the serving task has been lost because this will indefinitely suspend the caller. Thus TASKING_ERROR is raised by the run-time system in the calling task. This frees the calling task and allows it to prepare itself to use the alternate server.

Note that the server does not need to be aware that the caller has been replaced by an alternate if the caller's machine fails because the rendezvous is asymmetric. The entries in the server can be called by any task; in particular both the caller and the alternate caller.

If a rendezvous is not in progress when the failure occurs then processing on the remaining processor can continue. If this processor is executing the caller, then the caller will receive TASKING_ERROR the next time it attempts to rendezvous with the server and it will be reconfigured at that time. The alternate server will have already been started by then.

```
--
--    Code Resident On CPU 1.
--
task CALLER is
   entry TICK;
   -- TICK is an entry that is called
   -- periodically somehow and keeps the
   -- program synchronized in real time
end CALLER;

task body CALLER is
   type STATE  is (NORMAL, ABNORMAL);
   SYSTEM_STATE : STATE := NORMAL;
begin
   loop
      accept TICK;
      begin
         case SYSTEM_STATE is
           when NORMAL =>
             -- normal pre-rendezvous
             -- processing
             SERVER.E;
             -- normal post-rendezvous
             -- processing
           when ABNORMAL =>
             -- abnormal pre-rendezvous
             -- processing
             ALTERNATE_SERVER.E;
             -- abnormal post-rendezvous
             -- processing
         end case;
      exception
         when TASKING_ERROR=>
           SYSTEM_STATE := ABNORMAL;
           loop
             -- failure has occurred
             -- since reconfiguration may
             -- take time, "output"
             -- default values to keep
             -- physical devices happy
             OUTPUT_DEFAULTS;
             select
               -- rendezvous with the
               -- reconfiguration task
               -- to get data this task
               -- needs to operate
               RECONFIGURE_CPU_1.DATA(...);
               exit;
             or
               delay DELTA;
             end select;
           end loop
         when others =>
```

```
            -- handle other exceptions
         end;
      end loop;
   end CALLER;


   task ALTERNATE_SERVER is
      entry ABNORMAL_START(...);
      entry E;
   end ALTERNATE_SERVER;

   task body ALTERNATE_SERVER is
   begin
      accept ABNORMAL_START(...);
      loop
         -- pre-rendezvous processing
         accept E;
         -- post-rendezvous processing
      end loop;
   end ALTERNATE_SERVER;


   task RECONFIGURE_CPU_1 is
      entry CPU_2_FAIL;
      entry DATA(...);
   end RECONFIGURE_CPU_1;

   task body RECONFIGURE_CPU_1 is
   begin
      -- run-time system calls the following
      -- entry automatically when a failure
      -- of CPU 2 is detected
      accept CPU_2_FAIL do
         -- this call will start the alternate
         -- server on CPU 1 - the parameters
         -- will contain the data task needs
         ALTERNATE_SERVER.ABNORMAL_START(...);
      end accept;
      accept DATA(...) do
         -- prepare data for CALLER task
         -- when operating in the ABNORMAL
         -- system state
      end accept;
   end RECONFIGURE_CPU_1;


   --
   --     Code Resident On CPU 2.
   --
   task ALTERNATE_CALLER is
   entry ABNORMAL_START(...);
end ALTERNATE_CALLER;

task body ALTERNATE_CALLER IS
begin
   -- initialization code
   accept ABNORMAL_START(...);
   loop
      accept TICK;
      -- alternate processing
   end loop;
end ALTERNATE_CALLER;


task SERVER is
   entry E;
end SERVER;

task body SERVER is
begin
   loop
      -- pre-rendezvous processing
      accept E;
      -- post-rendezvous processing
   end loop;
end SERVER;


task RECONFIGURE_CPU_2 is
   entry CPU_1_FAIL;
end RECONFIGURE_CPU_2;

task body RECONFIGURE_CPU_2 is
begin
   -- run-time system calls the following
   -- entry automatically when a failure
   -- of CPU 1 is detected
   accept CPU_1_FAIL do
      -- this call will start the alternate
      -- caller on CPU 2 - the parameters
      -- will contain the data task needs
      ALTERNATE_CALLER.ABNORMAL_START(...);
   end accept;
end RECONFIGURE_CPU_2;
```

Conclusion

The execution of Ada programs on a distributed system is explicitly allowed but Ada does not make provision for processor failure in a distributed system. It can be argued that this is not part of the language's responsibility and that Ada should not address the issue. However, a distributed system that cannot cope with processor failure is no better than a uniprocessor system. Thus, we feel that the issue has to be dealt with somewhere and if it is not to be in the language, then the associated support environment can become unnecessarily complex.

The extended execution support system described in this paper can solve many of the problems. Combined with a careful programming style, distributed programs written in Ada which are tolerant to processor failures can be written. Many language details make the problem more difficult than it needs to be and attention to the issues of distributed processing in language design seems a desirable approach.

References

(1) Wensley, J. H. et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(2) Hopkins, A. L., et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(3) Tanenbaum, A. S., "Network Protocols", ACM Computing Surveys, Vol. 13, No. 4, December 1981.

(4) Reynolds P.F., J.C. Knight, J.I.A. Urquhart. "The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements", Final Report on NASA Grant No. NAG-1-260, NASA Langley Research Center, Hampton, Va.

(5) Programming Manual For The Ada Programming Language, U. S. Department of Defense, July 1982.

APPENDIX 3

Programming Language Requirements For Distributed Real-Time Systems Which Tolerate Processor Failure

J. C. Knight

J. I. A. Urquhart

Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville
Virginia, 22901
(804) 924-7201

ABSTRACT

In this paper, we discuss the programming of distributed systems that execute applications in which it is essential that continued service be provided after failure of some subset of the system's processors. We assume that the applications operate in real-time. The programming of such systems has usually been done on an ad hoc basis. Many different languages have been used; most were low-level providing few facilities for task communication, scheduling or reconfiguration. We suggest that it is essential that the programmer have control over the actions which occur following a failure. This implies that there must be facilities in the programming language to allow the programmer to specify his needs. In this paper we discuss deficiencies in existing languages, such as Ada+, and propose a set of requirements for languages to be used for programming crucial real-time applications on distributed systems.

+Ada is a trademark of the U. S. Department of Defense.

1. INTRODUCTION

One of the advantages of distributed processing is that a hardware failure need not remove all the computing facilities. If one processor fails, it is possible (at least in principle) for the others to continue to provide service. This fault-tolerant characteristic is very desirable for applications requiring high reliability. The use of distributed processing is further encouraged by the decreasing cost of computer hardware.

In this paper, we discuss distributed systems that execute crucial applications. By this we mean applications for which it is essential that continued service be provided after a failure. In general, responding to a failure by stopping the system and replacing the faulty component will not be acceptable.

A distributed system that is to be highly reliable will be built with a redundant bus structure. Redundancy usually includes replicating the bus along different routes as well as replication of the bus hardware itself on a particular route. Loss of a complete bus need be of little consequence if it is replicated and can be coped with by the low-level communications software. A complete break in the bus system that isolates some subset of the processors (i.e. the network becomes partitioned) is much more serious though very unlikely given multiple routes and replication. The issues that arise in that case are different from those arising from processor failure and will not be dealt with here. We consider only processor failures.

We assume that the applications operate in real-time. Thus a program may have to meet external deadlines, and success or failure of a program may depend on processor speeds and scheduling algorithms.

The programming of the kinds of systems we describe has usually been done on an ad hoc basis. Many different languages have been used; most were low-level providing few facilities for task communication, scheduling or reconfiguration. This was one of the situations that the Department of Defense sought to improve by the introduction of Ada [1]. Although Ada was carefully designed over several years with input from the entire computing community, recent work [2] has shown that it has serious deficiencies when used to program real-time distributed systems.

In a project to extend the language CLU to operate on distributed systems, Liskov [3] introduced the linguistic concept of guardians. That work addressed distributed systems where the nodes may be geographically remote and provide non-real-time service. The primary goal of guardians is the preservation of a system's database across failures in airline reservation and similar distributed systems. We are concerned with systems that operate in real time where provision of service is more important than preservation of data.

In this paper we discuss deficiencies in existing languages, such as Ada, and propose requirements for languages to be used for programming crucial real-time applications on distributed systems.

2. DISTRIBUTED SYSTEMS

The kind of architecture we expect to be in common use for real-time control systems in the future in shown in figure 1. It is based on the use of a high-performance data bus that links several processors. Each processor is equipped with its own memory. Devices such as displays, sensors, and actuators are connected to the bus via dedicated microprocessors. Thus these devices would be accessible from each processor. An example is a digital avionics system for a military aircraft. In these systems, separate computers may be used for flight control, navigation, displays, weapons management, and so on. The overall system requires some coordination and so the various computers communicate via a data bus. A typical system is described in [4].

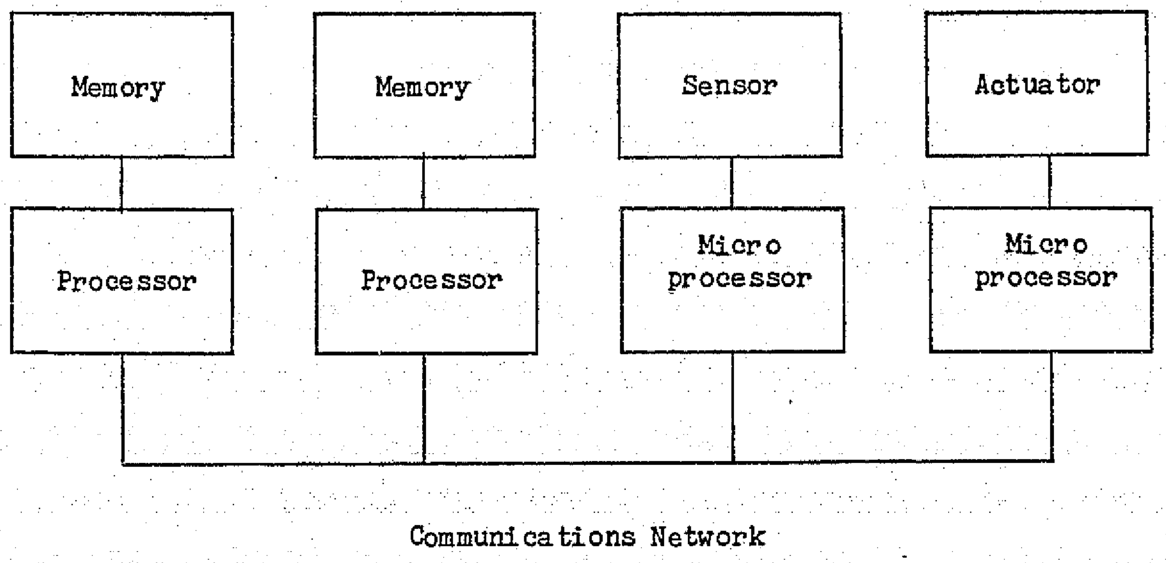


Figure 1 - Distributed Architecture

Much research has been undertaken in recent years to produce computer architectures of great reliability. There are, however, several reasons for employing software structures able to cope with partial hardware failure. Even though designed for reliability, any processor may still fail. Also, lightning, fire or physical damage could cause a processor to fail no matter how carefully the processor was built. At least with a distributed system there is the possibility that if part of the system was lost, what remained could continue to provide service.

A processor will be assumed to fail by stopping and remaining stopped. All data in the local memory of the processor will be assumed lost. Thus the case of a processor failing by continuing to process instructions in an incorrect manner and providing possibly incorrect data to other processors will not be considered. We assume that such events are taken care of by hardware checking within the processor. An alternative method using the Byzantine Generals algorithm is suggested by Schlichting and Schneider [5].

While this may seem a severe restriction, at least three arguments can be made in its favor:

(1) Faults of the assumed kind must be taken into consideration anyway since a processor might fail in this way.

(2) Either by hardware checking within a single processor or by checking between a dual pair of processors, it is possible for an underlying system to simulate the assumed processor failure mode.

(3) If such a failure mode is not assumed, error recovery becomes extremely difficult. It becomes possible for a processor to fail, and for the resulting errors to remain undetected until all data is compromised.

Given this assumption, error detection reduces to detecting that a processor has stopped. Error recovery is simplified by the knowledge that although data in the failed processor's memory is lost, data on the remaining processors is correct.

3. APPROACHES TO FAULT TOLERANCE

If a distributed system is to provide continued service after one or more processor failures, then facilities must be provided over and above those needed for normal service. We will refer to these as continuation facilities. If there is a single continuation facility for the entire system then the system is centralized. If the processor providing the continuation facility fails, the system stops and this is unacceptable. To prevent this, continuation facilities must exist on all the processors.

However, difficulties can still arise if, following the loss of a processor, a single continuation facility is chosen to perform fault tolerance for the entire system. For example, since the processor performing the fault tolerance may fail at any point, all other continuation facilities must be kept advised of the current state of the recovery so that they can take over if necessary. This is unacceptable and, in what follows, it will be assumed that each processor will have a continuation facility which independently assesses damage and effects whatever local changes are necessary for recovery in that processor.

On each processor remaining after a failure, the continuation facility must take the following actions:

Detect Failure

Some mechanism must detect processor failure and communicate its occurrence to the other parts of the continuation facility.

Assess Damage

Information must be provided so that a sensible choice of a response can be made. Certainly it must be known what processes were executing on the failed processor and what processes and processors remain. Further, in many applications the response will depend on other variables and these would also have to be known. The height of a aircraft, for example, might determine what actions should be taken when part of the avionics system is lost.

Processes executing on processors which survive the failure may still be affected by the failure. For example, their execution may depend on processes or contexts that were lost with the failed processor. If anything is to be done about such processes, they must be known and there must be some way of communicating with them.

Choose a Response

After a failure is reported to the software on a particular processor, the local continuation facility will independently decide on a response and put into effect any changes required on that processor. The choice of a response depends on a reconfiguration strategy and on information available. The information used might include both system information such as which processors are left and external information such as fuel level for example.

It is important that the information which the reconfiguration strategy uses be consistent across processors, since if it is not, the continuation facilities on different processors could decide on different responses and thus work at cross-purposes.

Effect the Response

Once a response has been decided on, it must be possible to carry it out. The continuation facilities should be able to abort processes, be able to communicate with processes so that the processes can take appropriate action on their own, and be able to start new processes. In many cases the new processes will have to be provided with data, and a consistent set of such data would have to be available to the continuation facilities.

Various difficulties are raised by the need for these actions. Firstly, two of the actions depend strongly on consistent data and without making quite unrealistic assumptions about the underlying message passing system, it cannot be assumed that data is consistent when a processor fails. A two phase protocol [6] can be used in this situation. In this protocol, a process executing on processor A sends copies of its data to processors B and C. B and C then each send A an acknowledgement but do not store the data in their consistent databases. When A has received acknowledgements from B and C it sends them commit messages. After receiving a commit message, B and C store the data in their consistent databases. It can be shown that with some additional processing [7] this allows processors in a distributed system to either all have copies of the new data or all know that old copies have to be used following a failure.

A second problem with the view of continuation taken above is the treatment of unrecoverable objects [8]. If an unrecoverable object has been modified, backward error recovery is not possible following a failure. The problem is no different on a distributed system than on a uniprocessor system. One apparent difference is that all the processors in a distributed system need to be informed of changes to unrecoverable objects and this has to be done in the presence of failures. This is actually a manifestation of the data consistency problem discussed above.

Given that tolerance to hardware faults is required, two completely different approaches can be considered. In the first approach, the loss of a processor is dealt with totally by the execution-time support software. Any processes which were lost are restarted on remaining processors, and all data is preserved by ensuring that multiple copies always exist in the memories of the various machines. We refer to this approach as _transparent_ since, in principle, the programmer is unaware of its existence. Transparent continuation has several advantages:

(1) The programmer need not be concerned with these aspects of fault tolerance.

(2) The programmer need not know about the distribution. Thus the distribution can be done by the system.

(3) The same program can be executed on different systems with different distributions.

However, since the continuation of service is transparent to the programmer, the programmer cannot specify degraded or 'safe' [9] service to be used following processor failure. The system cannot specify it either, and so transparent continuation must always provide identical service. If identical service is impossible, the system stops.

In crucial systems this is not acceptable. Situations will occur where identical service cannot be provided (due to physical damage, say) and yet degraded service is essential if some catastrophe is to be avoided. For example, a nuclear power plant may be unable to provide power but nonetheless must be able to shut down safely.

There are also many technical difficulties in implementing transparent continued service. Since failures can occur at arbitrary times, the support software must always be ready to reconfigure. Duplicate code must exist on all machines and up-to-date copies of data must always be available on all machines. The overhead involved in ensuring that all data is consistent on all machines all the time will be substantial. Even if transparent continuation could be offered without massive duplication of computing resources, it would be rejected for many applications because of its inability to offer alternate service

In the second approach to dealing with the loss of a processor only minimal facilities are provided by the execution-support software. The fact that equipment has been lost is made known to the program and it is expected to deal with the situation. We refer to this approach as programmer-controlled or non-transparent.

Programmer-controlled continuation has several disadvantages:

(1) The programmer must be concerned with all aspects of fault tolerance.

(2) The programmer must either specify the distribution or be prepared to deal with any distribution provided by the system.

(3) The program depends on the hardware system; at least the fault tolerant parts do.

The disadvantages are out-weighed by the fact that the service provided following failure need not be identical to the service provided before failure. Alternate, degraded service or 'safe' service can be offered if circumstances so dictate. In what follows only the non-transparent approach will be considered. We assume that the actions to be taken by each processor following a failure are specified within the software executing on each processor.

4. REQUIRED LANGUAGE SUPPORT

As in other fault-tolerant situations, fault tolerance in a distributed system requires facilities to allow failure to be detected, damage assessed, a response chosen and recovery effected. In addition however, consistent data is required across all processors so that the facilities may be distributed, and work towards the same end.

4.1. Source of Facilities

The necessary continuation facilities can be provided in different ways:

(1) By using mechanisms in the programming lánguage specifically designed for that purpose.

(2) By using mechanisms in the programming language which were designed for another purpose. If this were done, it would be a coincidence if the mechanisms worked satisfactorily since they were not designed to support fault tolerance.

(3) By using mechanisms outside the programming language such as modifications to the execution-time environment or software written in some other language, perhaps an assembly language.

The usual approach is the third where nothing specific is provided in the programming language. Such an approach is reasonable when only transparent fault tolerance is offered since in that case all facilities will be provided by the underlying system. However, it is not suitable for non-transparent fault-tolerance. If nothing is provided in the language, each system will develop its own methods. Programs will be non-portable and difficult to write and maintain. For example, if more data is required for a new reconfiguration strategy, and the part of the system that distributes data has been written in a way that is highly dependent on the particular data collected, then that part of the system will probably have to be rewritten.

The need to provide non-transparent fault tolerance leads to a general programming language requirement:

> A programming language which is to be used to program distributed systems must provide mechanisms to allow programs to be written which can cope with processor failure.

4.2. Fault Detection

The first facility that is required is detection of the loss of a processor. This can be done in several ways and may require hardware support or execution-time system support. However, it is clear that the applications' software will have to be informed of the failure so that the necessary actions to tolerate the failure can be initiated. Another requirement is therefore:

> A programming language which is to be used to program distributed systems must provide an interface to the underlying system to allow failure of processors to be signaled in a timely and unambiguous way.

4.3. Distribution of Objects

If the programmer is to specify algorithms which will be used for reconfiguration, it is essential that the programmer also specify distribution. If not the programmer must be prepared to deal with _any_ configuration chosen by the system. In a crucial system, this is not acceptable. For example, the system may choose to place both primary and alternate software for some service on the same machine.

The question of what to distribute then arises. It is clear that distribution must be relatively coarse-grained (tasks or packages in Ada, for example) or the job of specifying reconfiguration will become too difficult. A fundamental question is whether or not there should be units defined in the language specifically for distribution. A distribution unit, for example, could be restricted to use only local variables. Thus a third programming language requirement is:

> A programming language that is to be used to program distributed systems must specify what units can be distributed, how the distribution is to be done, and must include a syntax for expressing distribution.

Various problems arise if this requirement is not met. For example, an implementation may distribute five similar processes by providing a _single_ copy of the code (to save space) and arranging for it to be transported in blocks as needed. The specification of distribution may have been suitably stated but the implementation of the distribution has not been adequately defined. Failure of the machine holding the code will cause failure of the other processes which have been executing in machines which did not fail.

C-2

4.4. Object Dependence

In block structured languages, a program unit can assume the existence of an instance of all objects in the surrounding lexical blocks. When a system is distributed, it is possible to have a given program unit on one processor and one of its surrounding lexical blocks on another processor. If the latter processor fails, we must decide what to do with the surviving inner program unit.

In most languages it is possible to create objects that survive their surrounding contexts. Usually this happens when objects are created by some form of dynamic storage allocation. In Ada, for example, such objects exist until the context which contains the definition of the access type is left.

A first possibility is to treat surviving inner program units by applying rules of survival which already exist in the language. Alternatively new rules could be made for survival when a context is lost by processor failure, or the problem could be passed on to the programmer, who would have to specify survival conditions for each distributed unit.

The first possibility seems simplest, since it involves just using the language as usual. Unfortunately if, as is typical, everything depends on a 'main' process, failure of the processor executing the 'main' process will result in everything stopping. In Ada, for example, all the tasks defined in a main program depend on the main program. Thus, in fact, such systems are centralized as a result of the lexical structure of the language.

A further complication is that a program text describes a compile-time structure. Usually, no distinction is made between objects which exist at execution-time and objects which exist only at compile time. In a distributed system such a distinction is of the utmost importance. Unfortunately which objects are compile-time and which execution-time depends on the implementation. Under most implementations a program unit which uses a type definition from a surrounding context bears a completely different relationship to that context than a program unit which uses a variable defined in the context. If, for example, a unit uses a type definition in a surrounding scope, the unit would depend on the surrounding scope at compile-time but not at execution-time. In effect the type definition could be given a unique name and be copied into the unit. The unit could then survive the loss of the surrounding context. Similarly a program unit may (by the rules of the language) depend on another unit which will not exist at execution-time. An Ada library package containing only type definitions may be the master of an Ada task for example.

This leads to the following programming language requirement:

> A programming language which is to be used to program distributed systems must distinguish between compile-time and execution-time objects, and define survival rules for distributed program units.

4.5. Process Communication

Communication between processes on different processors is risky. For example, messages might be lost so that they must be re-sent and communication time becomes long, or one of the communicating processes might be on a processor that fails. In a crucial system, unless an arbitrarily long wait is acceptable at each communication, a process will need some way of withdrawing from every communication. Thus, another programming language requirement is:

> It should be possible to ensure that a process can meet a deadline no matter what happens to processes with which it is communicating. Failure of a processor should never cause a process executing on a surviving processor to be trapped while communicating.

While a time-out mechanism could deal with both slow response and lack of response because the responder is on a failed processor, a better scheme is to deal with the latter case differently. When a processor fails, knowledge of the failure becomes available to each machine. A process waiting for a response from a process on the failed processor can be dealt with in several ways by the continuation facilities:

(1) The waiting task could be aborted.

(2) The waiting task could be given information about the current state and allowed to decide on its own course of action.

(3) The communication from the waiting process could be switched to the replacement for the process which was lost.

The choice of what to do is part of the reconfiguration strategy. Subsequent attempts to communicate with the process which has disappeared could be treated in a similar way. Essentially the choices are:

(1) Replace the lost process by an identical substitute and transparently re-route calls.

(2) Replace the lost process by a substitute and inform callers.

(3) Do not replace the lost process and inform callers.

As before, the choice depends on the reconfiguration strategy. Note that the above requires that the continuation facilities know which processes are communicating, that is some form of communications log is necessary on each processor. This leads to the following programming language requirement:

> A programming language which is to be used to program distributed systems must provide facilities to deal with the loss of a communicating process. In particular, knowledge of which processes were communicating with the lost process and an ability to redirect communication may be needed.

4.6. Consistent Data

It is clear that some consistent data will always be required by the continuation facilities on each processor. At a minimum, when a processor fails, all continuation facilities must come to the same conclusion about which processes have been lost. In general, consistent data will also be required to restart processes or to start replacement processes.

Rather than have facilities for providing consistent data available to each applications' programmer, we believe that such a facility should be provided by the programming language. One possible scheme would provide a process-to-processor map for use by continuation facilities, and allow a programmer to specify data that was to be distributed to a consistent database in all processors. The language requirement is:

> A programming language that is to be used to program distributed systems must provide a mechanism to allow consistent data to be distributed across processors.

5. CONCLUSIONS

Almost all languages allowing processes to execute in parallel have taken the view that processes sharing a uniprocessor or executing on separate processors are logically equivalent. The program is taken to express the same algorithm in either case and the programmer need not be concerned with the actual implementation under which the program executes.

This is a convenient view so long as it is assumed either that processors will not fail or that a satisfactory response to failure is for the program to stop. If continued service after failure is desired, then the uni-processor and the multi-processor models are fundamentally different.

It is possible to shield the programmer from this difference by presenting the illusion of processors which do not fail. This is what we have described as transparent fault tolerance. As has been pointed out, the problem with transparent fault tolerance is that if the programmer views his program as executing on a very reliable uniprocessor he can have no concept of degraded or safe service. The system is either providing full service or no service.

The ability to provide alternate service is so important that a non-transparent approach must be taken where the programmer has to deal with reconfiguration. Existing high-level languages provide little or no aid for this; everything must be provided by the execution-time system, usually in a way that is strongly dependent on a particular implementation.

We have suggested a set of requirements for programming languages which are to be used to program crucial applications that execute on distributed systems. These requirements derive from the need to be able to continue to provide service after some subset of the available processors has been removed by unanticipated events. If services cannot be continued after a processor failure, then, in terms of reliability, a distributed system is no better than a uniprocessor.

A final comment, in some related work [2], we have examined Ada in the light of these requirements. In its present form it satisfies very few of them.

REFERENCES

(1) Reference Manual For The Ada Programming Language, U. S. Department of Defense, 1983.

(2) Knight, J. C., and J.I. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings Of The AIAA Computers In Aerospace Conference, Hartford, Connecticut, October 1983.

(3) Liskov, B., "On Linguistic Support For Distributed Programs", IEEE Transactions On Software Engineering, Vol. SE-8, No. 3, 1982.

(4) McTigue, T. V., "F/A-18 Software Development - A Case Study", Proceedings Of The AGARD Conference On Software For Avionics, The Hague, Netherlands, September 1982.

(5) Schlichting R. D., and F. B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", ACM Transactions On Computer Systems. Vol. 1, No. 3, 1983.

(6) Alsberg, P. A., and J. D. Day, "A Principle For Resilient Sharing Of Distributed Resources", Proceedings Of The International Conference On Software Engineering, San Francisco, October 1976.

(7) Gray, J. N., "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, New York 1978.

(8) Lee, P. A., "A Reconsideration Of The Recovery Block Scheme", Computer Journal, Vol. 21, No. 4, 1978.

(9) Leveson, N. G., and P. R. Harvey, "Analyzing Software Safety", IEEE Transactions On Software Engineering, Vol. SE-9, No. 5, 1983.

APPENDIX 4

A Testbed for Evaluating Fault-Tolerant Distributed Systems

John C. Knight

Samuel T. Gregory

Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville
Virginia, U.S.A. 22901
(804) 924-7201

ABSTRACT

When a strategy is proposed for enabling distributed software to tolerate hardware faults, its adequacy should be demonstrated experimentally in a scientifically believable way. Simply building and executing a program employing that strategy is not a scientifically believable demonstration. The difficulty lies in the inherent concurrency of distributed programs. It is unlikely that a situation that reveals a deficiency of the strategy will occur during a functional test. What is needed is an experimental testbed which can model any network topology and any software strategy designed to provide fault-tolerance. Also, it must allow any achievable software state to be precisely established, allow arbitrary parts of the distributed system to be failed, and allow the software to continue from that situation, if it can. Detailed requirements of such a testbed are given, and an implementation is described.

1. Introduction

We have been examining ways of providing tolerance to hardware faults in distributed programs written in Ada* [1]. We have suggested a strategy that we claim enables an application program to detect, survive and recover from failure of one or more of the machines of the system2 Given this proposal, we needed a testbed with which to validate or find difficulties in the claims of the proposed strategy. A typical question which has been asked about the strategy is: "What will happen in an Ada program if two tasks wish to rendezvous and the machine executing the accepting task fails _after_ the caller has sent a rendezvous request message but _before_ the accepting task has examined its entry queue? Will the proposed strategy actually handle this situation? " Another question involves task creation. The semantics of task creation seem to require that the task executing the unit declaring the new task wait for the new task's creation to complete. If the machine on which the new task is to execute fails during the task creation, it is necessary to demonstrate that the strategy is able to extricate the declaring task.

In the remainder of this paper, the problem is generalized from the verification of our Ada strategy, and the problem is analyzed for the requirements of a solution. The testbed which is our solution is then described.

*Ada is a trademark of the US DoD (AJPO).

2. The Problem

When a strategy is proposed for enabling distributed software to tolerate hardware failures, the adequacy of that strategy should be demonstrated experimentally in a scientifically believable way. Although there are exceptions, in practice this is rarely done. Instead, informal logical arguments are often given in an attempt to show that all cases have been considered or experiments are carried out in which failures are deliberately introduced. These failures are usually either random, or determined by the _external functions_ that the system is to provide. The intent is to answer questions about failures as they relate to the required external system behavior. For example, will the fault-tolerance strategy be able to cope with loss of processor(i) just after stimulus(j) is received.

Although fault tolerance strategies are well thought out, like any software design, they may contain weak points. Simply building and executing an instance of a program employing the proposed strategy is not a scientifically believable demonstration. The difficulty lies in the inherent concurrency of distributed programs. It is unlikely that a situation that reveals a deficiency of the strategy will occur during an operational test. What is needed is a testbed that allows such unlikely situations to be precisely established, and then allows observation of the software as it attempts to continue from that situation. A comprehensive set of such tests constitutes a scientific demonstration.

This problem extends beyond fault tolerance into the general area of semantic definitions of programming languages. Where a distributed system is to be built using a high-level language, it is often difficult to determine what a correct program is supposed to do even in the absence of faults. Even when semantic definitions of languages exist, and implementations are faithful to the definitions, the definitions tend to become vague with respect to the creation, deletion, and especially communications of processes. Where the definitions are less vague, they are often phrased in terms of intricate yet instantaneous operations. For example, the conditional entry call in Ada involves a check of the entry queues of the accepting task and of its ability to begin the rendezvous "immediately". For a system implemented on a single processor, the execution-time support for a language may achieve effects that appear to be instantaneous by disabling interrupts during most of, or all of, the operation. This prevents interference from other processes. On a distributed system, total suspension of parallel activity is either not possible or not permissible.

AdaED [3], New York University's validated implementation of Ada, has been suggested as an operational semantic definition of Ada. However, since that system provides no user control over task dispatching, it is not possible to answer "what if..." questions about Ada tasking semantics by executing programs using AdaED. Worse still, the execution of a particular tasking program on that system tells us nothing about its execution on distributed hardware. Again, what is needed is a testbed with which to set up the unusual circumstance corresponding to the "what if..." question, and allow the program or operational semantic definition to continue from that point, if it can.

3. The Requirements of the Solution

A distributed system is concurrent because parts of a distributed system execute on separate computers communicating in real time. One cannot test the parts separately and draw reasonable conclusions, so any experiments with it _must_ be done in a concurrent environment. A useful experimental testbed must be able to model any network topology and implement any software strategy designed to provide fault-tolerance. Also, it must be possible to deliberately fail _arbitrary_ parts of a distributed system under test when its software is in _any_ achievable state.

For sequential programs, methods have been available for a long time for setting up a desired program state and machine state combination, and stepping the program through its handling of the situation. These methods have been implemented in programs variously called "simulators" or "interpreters". In a simulator, the desired state is achieved by executing or single-stepping the program being tested to a breakpoint, or by forcing a particular value into the simulated program counter. Similarly, the desired machine state is achieved by instructing the simulator to force desired values into simulated registers or memory locations. Once the experiment is set up, results are usually obtained by single-stepping the subject program from that point while displaying the simulated machine state at each step.

A testbed which is analogous to a sequential program simulator, but which can simulate distributed (parallel) programs and can test fault-tolerance strategies, is required to meet the stated problem.

Specifically, the testbed must meet the following criteria:

(1) It must be able to model an arbitrary physical architecture. That is any number of processors in any network topology. This is not to imply that its capabilities be infinite, but that there should not be small and arbitrary hard limits, nor should only one or two configurations of processors be supportable.

(2) The testbed should provide at least the illusion (to the program executing on the processors) of parallel execution. It must be able to present a distributed software system with all of the problems the system will actually encounter on separate machines. For example, one of the problems is the inability of the fault-tolerance strategy to avoid interference by temporarily suspending execution of all but one process.

(3) Arbitrary logical architectures must be representable. The assignment of processes to processors must not be constrained by the design of the testbed. Any such constraint would limit the usefulness of the testbed in that it would prohibit many fault-tolerance strategies from being exercised on it.

(4) The testbed should be able to control inter-processor communication. It must allow the experimenter to introduce failures in the communications part of the modeled system, and to enforce the implications of the simulated processor topology. This involves maintaining the visibility and accessibility of messages anywhere in the system. To accompany this control, the level of instructions simulated should not be such that the issuance of

multiple messages is combined.

(5) In order to set up the precise situations desired for experiments, the testbed must provide explicit control over process execution. The experimenter must be able to stop a process on a specific instruction (i.e. a breakpoint), and to single-step instruction execution for particular processes. Since an experiment may entail arranging for many processes to be in many different states, control over suspension and release of one process should not be dependent upon control over suspension and release of any other process.

(6) Since a major point of the testbed is to see if software strategies can tolerate processor failures, the experimenter must be provided with the ability to fail and to restart processors at any desired point. The ability to restart is important because many strategies call for automatic replacement of defective hardware.

(7) The testbed must maintain simulated time. Since distributed systems often deal with the concept of time, the testbed's actions must not violate the abstracted machines' simulated clocks. For instance, the fact that any or all virtual processes "executing" on a particular abstracted machine might be held at breakpoints should have no effect on the relative progress of simulated time in the various other abstracted machines.

(8) The testbed must provide means of monitoring whether the fault-tolerant strategy under test worked or not. Since such results may manifest themselves in subtle ways, the entirety of the states of

the simulated processes and processors must be able to be displayed for examination. For the same reason, the displays should be selectable so as not to hide in a mass of irrelevant details those results considered important for a particular experiment.

These requirements have several implications. The testbed must maintain or represent a virtual state for each process in the distributed system being simulated. It must also maintain some minimal state information for each processor of the simulated system, if only that processor's idea of "current" time and which processes are considered to be executing on that processor. This helps both in organizing information for display, and in establishing loci of control over process execution and over inter-processor communication and processor failure. As a consequence of requirements five and seven, the testbed must be able to affect the simulated system's process scheduling algorithm without violating that algorithm's requirements. Each process can be brought to the desired state by executing to a breakpoint set for that process, and single-stepping for fine adjustment from there. The sequential simulator method of placing an arbitrary value into the process's instruction pointer to bring it to a desired state could cause invalid results from experiments, since that might prohibit the fault-tolerance strategy from gathering, along the way, whatever information it needed for a proper recovery. In short, the testbed should not allow the experimenter to set up truly impossible combinations of process and system states, but it should allow every other combination to be reached and held as a point for the introduction of a failure.

4. The Implementation

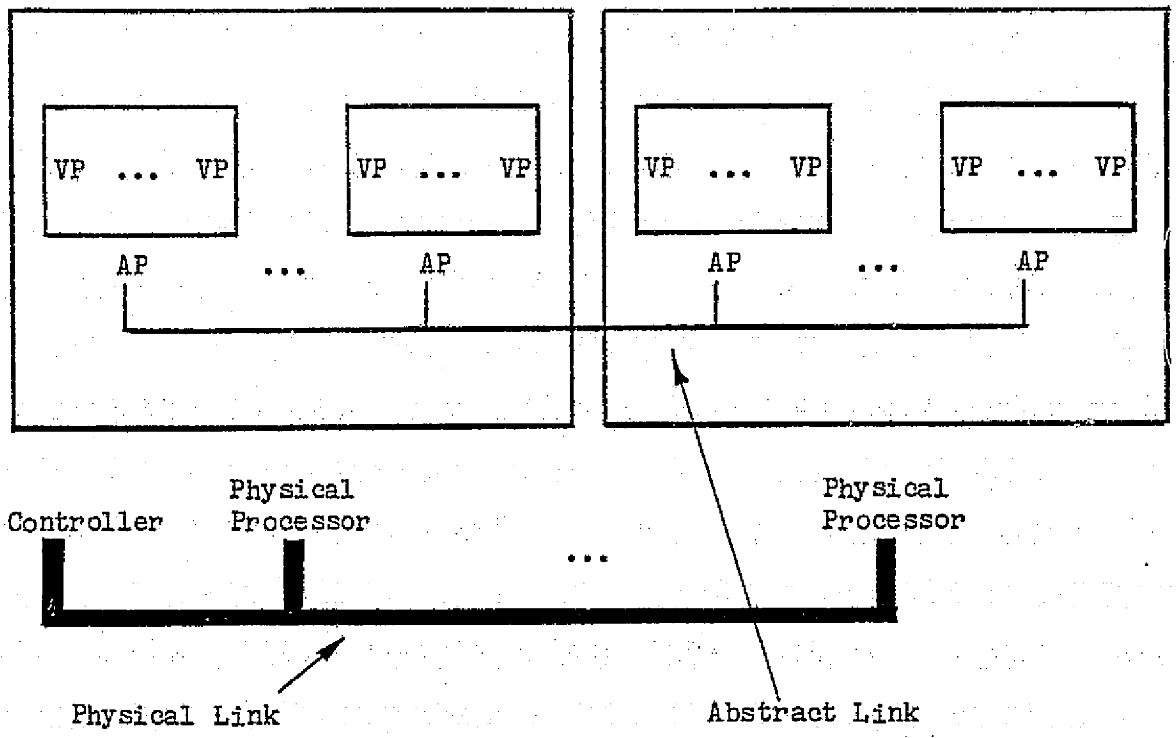
The motivation behind this distributed system testbed lies in the desire to validate a proposed strategy for having distributed Ada programs survive machine failures [2]. The fault-tolerance strategy is expressed in Ada, and the testbed we are constructing has an interpreter of an intermediate code at the individual message level.

4.1. Overview

The organization of the testbed is illustrated in . The part of the testbed which supports the execution of individual Ada processes in a system under test is the set of virtual processors. The term comes from the operating system concept that every process in a system is to have the illusion that it is executing on its own processor with an instruction set enhanced by special "instructions" usually known as supervisor calls.

The testbed provides a user-specifiable number of processors referred to as abstract processors. Each is capable of multiprocessing the execution of from zero to all processes (i.e. virtual processors) in the system to be simulated. An abstract processor employs a user-supplied process scheduling algorithm which defaults to a round-robin scheme. A system being tested is intended to view an abstract processor as an actual machine with some execution-time support code providing the multiprogramming illusion. Thus a complete system under test will consist of a set of abstract processors connected by an abstract communications facility.

The correspondence between <u>physical processors</u> and abstract processors is similar to that between abstract processors and virtual processors. Each physical processor is capable of executing from zero to all possible abstract processors, with the distribution controlled by an experimenter-supplied map. A physical processor multiprocesses a group of abstract processors, using a fair scheduling algorithm which,



VP - Virtual Processor
AP - Abstract Processor

"Fault-Tolerant Distributed System Testbed"

Figure 1.

again, can be altered by the experimenter. One physical processor is called the controller and executes the command interpreter which serves as a user interface for the experimenter. All other physical processors provide an underlying portability structure for the testbed as will be described below. Note that physical processors are not necessarily real processors.

4.2. Abstract Processors

The instruction set that an abstract processor provides its processes is easily modifiable, and both a "crystal" frequency and instruction and message "execution" times for automatically updating the abstract processor's clock are accessible to the experimenter. The abstract processor also provides a place for user code which may modify message traffic, if that should be part of a fault-tolerance strategy to be tested. Thus, the abstract processor part of the testbed represents what would typically be found in both the processing hardware and the execution-time support code and "system level" code of the distributed system to be simulated.

4.3. The Message Mechanism

The abstract processors of the testbed communicate via messages. The experimenter also communicates with the abstract processors and other areas of the testbed via these messages. A message is represented as a Pascal variant record, making it easy for the experimenter to add any new messages required for communication among a particular fault-tolerance strategy's processes. As mentioned, the experimenter is able to insert code to implement a particular strategy's modifications to message traffic, if required. The experimenter can also dynamically influence the delivery of messages from any source (such as the Ada program or its support). The default network topology for abstract processors is modeled after Ethernet.

The message handling facilities also aid the abstract processors in enforcing time. An abstract processor removes messages from the set of messages destined for it (or its processes) in a variation of first-in-first-out order. For any particular retrieval, no message which is time-stamped in the future (according to that abstract processor's clock) will be retrieved. An abstract processor's clock is essentially its view of wall clock time. We will say more on this later. This allows abstract processors to be truly executed in parallel. One abstract processor can have advanced its clock well beyond that of another, but any message sent from the first to the second is guaranteed to "arrive" after the time of its time-stamp in the view of the receiver.

4.4. Virtual Processors

In our problem, there is a one-to-one correspondence between Ada tasks and virtual processors. A virtual processor implements a synthetic instruction set which can be used as a target by an Ada translator. Thus each Ada task is "executed" by the fetch-execute cycle of a virtual processor.

As mentioned, many virtual processors may be supported by any one abstract processor. Thus an abstract processor provides the same multiprogramming illusion that is normally provided by a language's execution-time support or an operating system. The form of the virtual processor desired for a particular system to be tested is expected to vary widely. Hence, the instruction set is easily modifiable. Its present form is designed only to support Ada.

A virtual processor's state is represented as a Pascal record which is accessible to the experimenter. As well as the data structures necessary to support its instruction set, the virtual processor's state contains the accounting and scheduling information used by the abstract processor in supporting the multiprogramming illusion. For our problem, the virtual processor's state includes information which is updated during interpretation of certain instructions and messages, and employed by the scheduling algorithm. It also contains the task's stack of local variables (referencing environments) and its list of dependent tasks, and so on.

The style of the instruction set we have implemented in the virtual processors elucidates the requirement concerning the level of instructions simulated. The instruction set would in normal circumstances be called an intermediate code. Each instruction is at a very high level, corresponding in many cases to long sequences of instructions on actual computers. This allows us to combine and forget details considered uninteresting to our particular project. On the other hand, we are interested in determining whether our proposed fault-tolerance strategy can recover from the loss of processors during an exchange of messages by tasks residing on separate processors. Thus, rather than grouping the issuance of a message, the handling of the response, and issuance of any reply into a single supervisor call instruction, we provide several supervisor call instructions which are always used together and in the same order. This allows the experimenter to hold a virtual processor at any interesting point within the message exchange while he deliberately fails an abstract processor.

4.5. Experiment Control

During an experiment, the experimenter can set and remove breakpoints for individual virtual processors (i.e. Ada tasks), can direct that an abstract processor's crystal frequency be increased or decreased, can direct that an abstract processor cease to exist (introduction of a machine failure), or be restored (simulate standby spares), can single-step or execute any one or a group of virtual processors, and can alter delay intervals in message transmissions (for different processor configurations). This is a small number of capabilities, but it turns out to be all that is really needed. The experimenter could easily be allowed to alter simulated memory locations as in a sequential simulator, but hardware redundancy technologies seem to have that kind of hardware fault under control making it uninteresting to software strategies for tolerating hardware faults.

4.6. Simulated Time

The speed of each abstract processor is adjustable and a clock which tells that abstract processor's view of wall clock time is maintained. Each instruction and each message of the system under test has associated with it the number of machine cycles needed for its processing. As instructions are executed and messages are handled, the appropriate abstract processor's clock is incremented by the product of its speed and the appropriate machine cycle count. An abstract processor which has nothing to do, such as is the case when all of its virtual processors are held at breakpoints, continues to be scheduled and its clock is adjusted by a minimal amount. This prevents conflicts between the artificial suspension of time by breakpoints and the previously-mentioned refusal of an abstract processor to accept a message time-stamped in what it considers to be the future.

The algorithm scheduling virtual processors can no more violate accurate simulation of time than it could violate wall clock time if used in an actual distributed implementation. A bad scheduling algorithm in the system under test would be equally bad in actual operation, so the testbed should not be responsible for "correcting" it. The scheduling of abstract processors within the testbed, however, could violate time. For example, suppose there are three abstract processors each executing a virtual processor and that the virtual processors wish to communicate with each other. Further, let the communications be initiated by the virtual processors on abstract processors 1 and 2 and be directed toward the virtual processor on abstract processor 3. A simple round-robin scheduling scheme would always bias the load of pending messages for abstract processor 3. Whenever abstract processor 3 was scheduled, it would have a backlog of pending messages, the handling of which might use up all of its allotted time, causing it never to make progress in executing its virtual processors. Our implementation of the testbed avoids this problem of message biases violating simulated time by providing a particular (default) abstract processor scheduling algorithm. A random choice without replacement is made from available abstract processors, replacing all of them into the scheduling pool only when it becomes empty. As will be seen, our implementation is also capable of providing actual parallel execution of the abstract processors.

4.7. Physical Realization

The testbed's lowest communication layers are modular so that, a simple substitution and re-compilation allows the testbed to execute either on a single minicomputer running UNIX* or on an actual distributed system consisting of a set of IBM Personal Computers linked together via Ethernet. When executing on the minicomputer, a set of UNIX processes are created which correspond to the IBM Personal Computers and their associated software. The processes communicate using "pipes" and these correspond to the Ethernet. That part of the testbed executing as a UNIX process or on its own Personal Computer is a physical processor. The experimenter's interface, called the controller, is a physical processor and communicates with the rest of the physical processors, and hence the abstract processors and virtual processors, via the same pathways as they use to communicate with each other. In all areas of the testbed, preference is given to dealing with the controller's messages, so the experimenter's commands take effect immediately.

The separation of physical processor and abstract processor levels allows the testbed to be executed on a single processor without a

*UNIX is a trademark of Bell Laboratories.

multitasking operating system if that is the limit of one's environment. However, the testbed can be configured with a single abstract processor implemented on each physical processor, and, when executing on the IBM Personal Computers, the testbed then implements a true distributed system.

4.8. Displays

Each of the physical processors has a monitor associated with it. If a physical processor is a UNIX process, it has its own terminal at which the experimenter may select displays of simulated activities. If a physical processor is a program on a Personal Computer, the Personal Computer's keyboard and monitor serve in the same capacity. The experimenter can choose among any of several displays to be continuously updated on the monitors. The displays include

(1) values output from the Ada program being tested,

(2) the state of any named task, including its entry queues,

(3) the remainder of the abstracted machine's state information,

(4) the contents of any queue or buffer of messages in that physical processor's part of the simulated system.

The controller monitors message traffic between the testbed's parts, so it is capable at any time of displaying all relevant breakpoints, and the mappings from:

(1) Ada source task names to virtual processor numbers,

(2) virtual processor numbers to abstract processors,

(3) abstract processors to physical processors.

4.9. Implementation Status

As of this writing, the structures implementing the physical processors, abstract processors, and virtual processors as described above, and the implementation of message traffic are all in place. A "friendly" user interface for the controller is being developed. Commands for altering network topology are not being included at the present time, because such concerns are not pertinent to our current problem of investigating the use of Ada on distributed systems. The default abstract processor scheduling algorithm will not be changed for our problem, and we do not need priorities for Ada tasks, so the default round-robin virtual processor scheduling is also sufficient.

The fault-tolerance strategy to be tested calls for code to manipulate and alter the sequences of messages between virtual processors at the abstract processor level, so we will be using that feature. That code is being built incrementally as the virtual processor message handler coding progresses. A preliminary version of the testbed that has three physical processors but with only a minimal implementation of the virtual processors, has been executed as three processes on a DEC VAX* 11/780 running UNIX. In those trials, the ability to fail abstract processors, to transport messages to their proper destinations, and to monitor various parts of the state data

*VAX is a trademark of Digital Equipment Corporation.

structures was demonstrated. We are in the process of moving the testbed to a set of IBM Personal Computers and extending the implementation of the virtual processors.

5. Conclusion

A testbed for evaluating fault-tolerant distributed systems has been described. It allows a system to be methodically tested, and allows observations to be made of its response to hardware failures with the software in any internal configuration.

The testbed allows quite general distributed systems to be tested. It is presently being used to evaluate an implementation of Ada in a distributed environment.

REFERENCES

(1) Reference Manual For The Ada Programming Language, U. S. Department of Defense, 1983.

(2) J. C. Knight and J. I. A. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings Of The AIAA Computers In Aerospace Conference, Hartford, Connecticut, October 1983.

(3) R. B. K. Dewar, G. A. Fisher, E. Schonberg, R. Froehlich, S. Bryant, C. F. Goss, and M. Burke, "The NYU Ada Translator and Interpreter," Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, November 1980.

APPENDIX 5

ON THE IMPLEMENTATION AND USE OF Ada ON FAULT-TOLERANT DISTRIBUTED SYSTEMS

John C. Knight

John I. A. Urquhart

Department of Applied Mathematics and Computer Science

University of Virginia

Charlottesville

Virginia, 22901

-- D R A F T --

I INTRODUCTION

Digital computers are being used increasingly in dedicated control applications that require high reliability. These systems are usually embedded and frequently distributed. Several processors may be used that communicate using a high-speed bus even though they are geographically close. An example is a digital avionics system for a military aircraft in which separate computers may be used for flight control, navigation, displays, weapons management, and so on. The overall system requires some coordination and so the various computers communicate via a data bus. A typical system is described by McTigue [1].

One of the advantages of distributed processing is that a hardware failure need not remove all the computing facilities. If one processor fails, it is possible (at least in principle) for the others to continue to provide service. This is a desirable characteristic for applications requiring high reliability. The use of distributed processing is further encouraged by the decreasing cost of computer hardware.

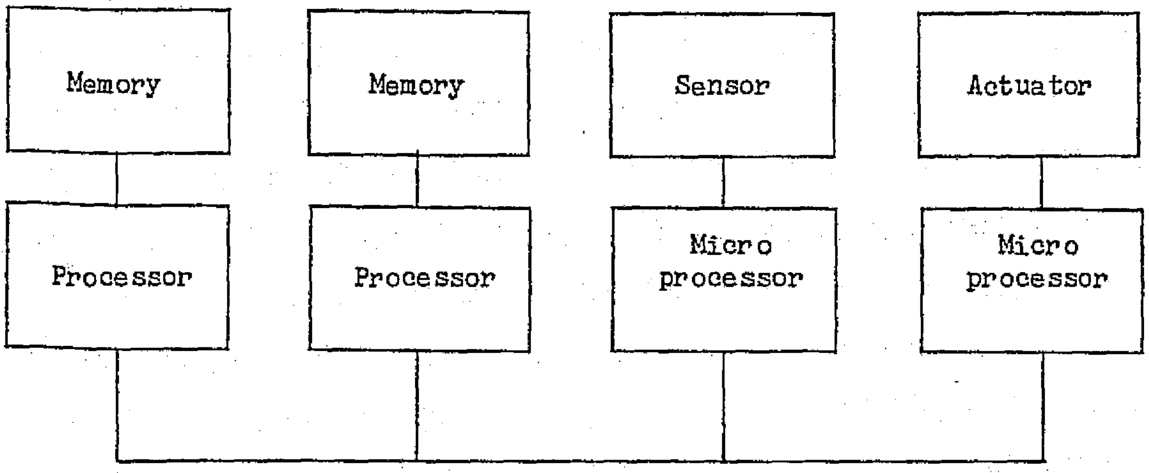
Ada [2] was designed for the programming of embedded systems (such as those mentioned above) and has many characteristics designed to promote the development of reliable software. In this paper we examine the problem of programming distributed systems in Ada. In particular, we are concerned with the issues that arise where some form of acceptable processing must be provided using the hardware facilities remaining after a failure.

In section II we present some motivation for considering distributed systems where hardware failure must be tolerated, and define in detail the failures we will consider. In section III we look at the general problem of providing service after processor failure, and we present the general facilities needed to support fault tolerance in section IV. The considerable difficulties that arise when such systems are programmed in Ada are discussed in section V. We show in sections VI and VII that these difficulties can be overcome by careful programming and by making extensive additions to the execution-time support system that would normally be needed to support Ada. These additions make no changes to the language itself and their use in Ada is discussed in section VIII. An example of an application program using such mechanisms is given in appendix 1, and a detailed examination of task communication difficulties is presented in appendix 2.

II HARDWARE TOPOLOGY AND FAILURES

The kind of architecture we expect to be in common use for embedded systems in the future in shown in figure 1. It is based on the use of a high-performance data bus that links several processors. Each processor is equipped with its own memory. Devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors. Thus these devices would be accessible from each processor.

A great deal of research has been undertaken in recent years to produce computer architectures of great reliability such as the SIFT [3] and FTMP [4] machines. However, even though designed for reliability,



Communications Network

Figure 1 - Distributed Architecture

these machines may still fail. Lightning, fire or other physical damage could cause a processor to fail no matter how carefully the processor was built. There is, therefore, good reason for employing software structures able to cope with partial hardware failure.

It is clear that continued service after failure implies a system distributed over two or more processors. Distribution, however, does not necessarily imply continued service after failure. Several processors sharing a computation that stops whenever any single processor fails will be viewed as a single processing unit. This distinction between distributed systems that allow continued service after failure and those that do not is important. We will call a distributed system that does not allow continued service after a processor fails a <u>centralized</u> distributed system. From the perspective of fault tolerance, such a system is no better than a uniprocessor.

We assume that communication between processors on a distributed system will be implemented using software that conforms for the most part to the lower layers of the ISO standard seven-layer Reference Model [5]. The kinds of hardware failure that we are concerned with are not addressed by the ISO protocol. The ISO protocol is concerned with communications failure such as dropped bits caused by noise, loss of messages or parts of messages, etc. The only class of faults not dealt with elsewhere is the total loss of a processor or a data bus with no warning.

A processor will be assumed to fail by stopping and remaining stopped. All data in the local memory of the processor will be assumed lost. Thus the case of a processor failing by continuing to process instructions in an incorrect manner and providing possibly incorrect data to other processors will not be considered. We assume that such events are taken care of by hardware checking within the processor or the methods of Schlichting and Schneider [6].

While this may seem a severe restriction, at least three arguments can be made in its favor:

(1) Faults of the assumed kind must be taken into consideration anyway since a processor might fail in this way.

(2) Either by hardware checking within a single processor or by checking between a dual pair of processors, it is possible for an underlying system to simulate the assumed processor failure mode.

(3) If such a failure mode is not assumed, error recovery becomes extremely difficult. It becomes possible for a processor to fail, and for resulting errors to remain undetected until all data is compromised.

Given this assumption, error detection reduces to detecting that a processor has stopped. Error recovery is simplified by the knowledge that although the data in the failed processor's memory is lost, data on the remaining processors is correct.

A distributed system that is to be highly reliable will be built with a redundant bus structure. Redundancy usually includes replicating the bus along different routes as well as replication of the bus hardware itself on a particular route. Loss of a complete bus need be of little consequence if it is replicated and can be coped with by the low-level communications software. A complete break in the bus system that isolates some subset of the processors (i.e. the network becomes partitioned) is much more serious though very unlikely given multiple routes and replication. The issues that arise in that case are different from those arising from processor failure. They are beyond the scope of this paper and will not be dealt with here.

III APPROACHES TO FAULT TOLERANCE

There are two completely different approaches that can be taken when attempting to provide tolerance to hardware faults. In the first approach, the loss of a processor is dealt with totally by the execution-time support software. Any services that were lost are assumed by remaining processors, and all data is preserved by ensuring that multiple copies always exist in the memories of the various machines. We describe this approach as _transparent_ since, in principle, the programmer is unaware of its existence. This is the approach being pursued by Honeywell [7].

Transparent continuation has several advantages:

(1) The programmer need not be concerned with reconfiguration.

(2) The programmer need not know about the distribution. Thus the distribution can be done by the system.

(3) The same program can be run on different systems with different distributions.

However, as the continuation of service is transparent to the programmer, the programmer cannot specify degraded or _safe_ [8] service to be used following processor failure. Since the system cannot specify it either, transparent continuation must always provide identical service. If identical service is impossible because insufficient processing resources remain, the system stops.

In many crucial systems this is not acceptable. Situations will occur where identical service cannot be provided (due to physical damage, say) and yet degraded service is necessary if some catastrophe is to be avoided. A nuclear power plant may be unable to provide power but nonetheless must be able to shut down safely.

There are also many difficulties in implementing transparent continued service. Since failures can occur at arbitrary times, the support software must always be ready to reconfigure. Duplicate code must exist on all machines and up-to-date copies of data must always be available on all machines. The overhead involved in this process is considerable. Further, the overhead may not be obvious to the programmer when the program is being written. A simple assignment statement, for example, may take considerable time to execute in order to ensure that the updated value of the variable has been distributed to all the memories. However, even if these difficulties could be overcome and transparent continuation could be offered without massive duplication of computing resources, it would still be unsuitable for many applications because of its inability to offer alternate service.

In the second approach to dealing with the loss of a processor only minimal facilities are provided by the execution-support software. The fact that equipment has been lost is made known to the program and it is expected to deal with the situation. We will refer to this approach as programmer-controlled or <u>non-transparent</u>.

Programmer-controlled continuation has several disadvantages:

(1) The programmer must be concerned with reconfiguration.

(2) The programmer must either specify the distribution or be prepared to deal with any distribution provided by the system.

(3) The program depends on the system; at least the reconfiguration parts do.

The disadvantages are out-weighed by the fact that the service provided following failure need not be identical to the service provided before failure. The programmer can have complete control over the services provided by the software and the actions taken by the software following failure. Alternate, degraded, or merely "safe" service can be offered as circumstances dictate. Also, the various inefficiencies associated with the handling of failure and the necessary preparations for it are quite clear to the programmer.

In the remainder of this paper we will consider only the non-transparent approach. We assume that the actions to be taken by each processor following a failure are specified within the software executing on that processor.

IV REQUIRED CONTINUATION FACILITIES

If a distributed system is to provide continued service after one or more processor failures, then facilities must be provided over and above those needed for normal service. We will refer to these as continuation facilities. If there is a single continuation facility for the entire system then the system is centralized. If the processor providing the continuation facility fails, the system stops and this is unacceptable. To prevent this, continuation facilities must exist on all the processors.

However, difficulties can still arise if, following the loss of a processor, a continuation facility on one processor is chosen to perform fault tolerance for the entire system. For example, since the processor performing the fault tolerance may fail at any point, all other continuation facilities must be kept advised of the current state of the recovery so that they can take over if necessary. This is unacceptable because of the resulting overhead. In what follows, we assume that each processor will have a continuation facility that independently assesses damage and effects whatever local changes are necessary for recovery in that processor.

Continued service after one or more processor failures requires that the following actions be performed:

(1) Detect Failure

Processor failure must be detected and communicated to the software on each of the remaining processors.

(2) Assess Damage

It must be known what processes were running on the failed processor, and what processes and processors remain. Further, processes executing on processors that survive the failure may still be affected by the failure. For example, their execution may depend on processes or contexts that were lost with the failed processor. If anything is to be done about such processes, they must be known and there must be some way of communicating with them.

(3) Select A Response

Information must be provided so that a sensible choice of a response can be made. The response that is chosen will be determined by which processors and processes remain, but in many applications the response will also be determined by other variables and these would also have to be known. The height of an aircraft, for example, might determine what actions should be taken when part of the avionics system is lost.

After a failure is reported to the software on a particular processor, the local reconfiguration software will independently decide on a response and put into effect the changes required on that processor. The choice of a response depends on the reconfiguration strategy provided by the programmer and on the information provided. It is important that the information which the reconfiguration strategy uses be consistent across processors, since if it is not, reconfiguration processes on different processors could decide on different responses and thus work at cross-purposes.

(4) Effect The Response

Once a response has been decided on, it must be possible to carry it out. The reconfiguration software should be able to create and remove processes, start and stop processes, and be able to communicate with processes so that they can take appropriate action on their own. In many cases the new processes will have to be provided with data, and a consistent set of such data would have to be available to the reconfiguration software.

Various difficulties are raised by these requirements. Firstly, they depend strongly on data that is consistent across all machines. Without making quite unrealistic assumptions about the underlying message passing system, it cannot be assumed that data is consistent when a processor fails if no precautions are taken. However, a two phase protocol [9,10] can be used in this situation to ensure that consistent data (but perhaps not the most recent values) is available on all machines.

A second problem with the view of continuation taken above is the treatment of unrecoverable objects [11]. If an unrecoverable object has been modified, backward error recovery is not possible following a failure. The problem is no different on a distributed system than on a uniprocessor system. An apparent difference is that all the processors in a distributed system need to be informed of changes to unrecoverable objects and this has to be done in the presence of failures. However, distribution of status information about unrecoverable objects is just an example of the data consistency problem discussed above.

V DISTRIBUTION AND CONTINUATION IN Ada

We now consider the use of Ada for programming distributed systems in which processor failure has to be tolerated. What is needed is a distributed system that provides the continuation facilities discussed in section IV.

Distribution

The choice of objects to be distributed is an important question in the design of a distributed system. Ada has a tasking mechanism and, according to the Ada Reference Manual [2], it is intended that tasks be distributed in an Ada program:

Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor.

Also, it is clear from the requirements for the language [12] and from the Ada Reference Manual [2] that the tasking facilities are intended to be used for all task communication and synchronization even when different physical processors are executing the tasks involved. While it would be possible to devise a separate mechanism for inter-task activities between computers using some form of input and output, this would be substantially less useful than existing facilities and probably program-specific.

No facilities are defined in Ada to control the distribution of tasks. It is essential that software that is to be used following the failure of a particular processor not be resident in the memory of that processor (otherwise the system would be a centralized). To achieve this separation, it is essential that the programmer be able to control the placement of both the primary and alternate software. Surprisingly, there is no explicit mechanism for control of distribution in Ada although there are representation clauses to control the bit-level layout of records, to allow the placement of objects at particular addresses within a memory, and to associate interrupt handlers with specific machine addresses.

It is not sufficient to be able to control the allocation of tasks to processors. The semantics of task distribution must take into account the possibility of failures. For example, if there are multiple tasks of a particular task type and they are executing on different processors, a separate copy of the code must be required for each processor. Otherwise, an implementation could provide a single copy of the code that was shared by all processors; for example by fetching a copy when a new task body is elaborated. This would be satisfactory if there were no failures. However, failure of the processor containing the original copy of the code would then suspend all subsequent elaborations.

Continuation

For any particular programming language, the required continuation facilities discussed in section IV could be provided in three different ways:

(1) By using mechanisms in the programming language specifically designed for that purpose.

(2) By using mechanisms in the programming language that were designed for another purpose. If this were done, it would be a coincidence if the mechanisms worked satisfactorily since they were not designed to support fault tolerance.

(3) By using mechanisms outside the programming language such as modifications to the execution-time environment or software written in some other language, perhaps an assembly language.

Unfortunately, Ada makes no explicit provision for continuation. Many features of the language raise substantial difficulties in damage assessment, and in selecting and effecting a response.

Failure Detection

An execution-support system for Ada is not required to provide any facilities for detection of processor failure. No specific interface is provided by the language to allow software to be informed of processor failure.

If failure could be detected, it might be possible to inform the software by raising an exception or generating an interrupt; the latter using an entry call as its interface. In either case, appropriate placement of the corresponding exception handler or accept statement becomes a problem since it must be assured that they will be executed when required. Also, the necessary exception and entry names are not predefined and so their use is neither standardized nor required.

Damage Assessment

Clearly, the damage sustained as a result of a processor failure includes loss of the services that were provided by the software that was executing on the processor that failed. It also includes loss of the data contained in the memory of the failed processor. In addition, in an Ada program the failure of a processor can cause damage to the software that remains. Broadly speaking, two forms of damage can occur. A task can be suspended waiting for a message that will never arrive, and a task can lose part of its context. These will be discussed in turn.

Task Communication

The problems that arise in task communication are best illustrated by an example. Consider an Ada program that contains two tasks A and B where A is executing on one processor and B on another. Suppose that task A has made a call to an entry in task B, and that B has started the corresponding rendezvous. If B's processor now fails, task A will remain suspended forever because the rendezvous will never end. Since the failure takes place after the start of the rendezvous, a timed or conditional entry call will not avoid the difficulty.

Similar problems arise throughout Ada, both in explicit communication such as the rendezvous and in implicit communications such as task activation. A detailed examination of these situations is given in appendix 2 and in [13].

Loss Of Context

In block structured languages, a program unit can assume the existence of an instance of all objects in the surrounding lexical blocks. When a system is distributed, it is possible to have a given program unit on one processor and one of its surrounding lexical blocks on another processor. If the latter processor fails, it must be decided what to do with the surviving inner program unit.

A task in Ada relies on several contexts:

(1) The context of the body. This is the lexical units enclosing the body of the task.

(2) The creator. This is the program unit which creates the task.

(3) The masters [21, page 9-4].

All of these contexts may be different. Each of them may be lost due to processor failure. Ada defines no semantics for these situations.

Thus, the damage following processor failure will include lost services, lost data, the permanent suspension of tasks on remaining processors for a variety of reasons, and the loss of contexts of some tasks. This damage could be quite extensive. As it is presently defined, Ada provides no way of determining the extent of this damage. Indeed, any attempt to assess the damage could cause the task enquiring about the damage to be suspended itself. Suppose for example that the attribute callable was used by a task to determine whether another task was callable; the interpretation of the value true being that the second task was still present and providing service. If the tasks were on separate machines, the implementation of the attribute would require an exchange of messages. Since no reply could ever be received after a failure, the enquiring task would effectively be suspended.

Selecting and Effecting The Response

The purpose of effecting a response is to replace services that were lost. The source of the new services will have to be software that resides on machines that remain after the failure.

Ada has facilities for starting tasks and for creating them. It is a relatively simple matter to cause software to begin execution to replace lost services. Note however that the replacement software is part of the same program as the software that was lost, and so cannot use names that would be ambiguous to a compiler. Thus for example a replacement task cannot be given the same name as the task it is intended to replace. Although tasks can be created and started, the scope rules of the language limit the lexical placement of the alternate software since it must be within the scope of the software effecting the response. Recall also that Ada does not provide the necessary distribution control.

Even if replacement services could be started and distribution could be controlled, it is still necessary for the replacement software to communicate with the software remaining after the failure. This means that communication paths used before the failure have to be redirected. Communication will be primarily by rendezvous. The rendezvous in Ada is asymmetric and so a calling task needs to know the name of any task containing an entry it wishes to call, but a called task need not know the names of tasks that will call it. If a calling task has to be replaced because of a failure, the replacement can call the same entry that was called by the lost task. The entry is still available in the same task that was being called before the failure. Thus redirection is trivial if a calling task is lost. However, if a called task has to be replaced because of a failure, the replacement cannot be given the same name as the task that was lost. This would duplicate the definition of a task name in the same scope. Thus, in this case, redirection is very involved. The replacement called task will have to have a different name and, more importantly, all the calling tasks (that may not have been replaced) will have to begin using a different name in their entry calls.

This difficulty is not quite so serious for tasks created by allocators. Since assignment is allowed for access variables, communication can be redirected by assigning a value representing an alternate task to an access variable used to make entry calls. Two problems then arise. First, the entire program has to be written using access variables to access tasks, and second, alternate tasks have to be of the same type as the primary task which may not be convenient.

Selecting and implementing a response relies on the availability of data that is consistent across machines. Ada makes no provision for ensuring that data can be reliably distributed across machines.

Finally, in effecting a response, it will be necessary to take care of those tasks damaged by the failure. The only way that this can be done in Ada is to abort them. Further, since some computing facilities have been lost, the response that is chosen might also involve modifying services that were not affected by the failure by aborting some tasks and starting others. Because of the scope rules of the language, aborting several tasks will be difficult to arrange if the program makes use of nesting and a single piece of software is to contain all the necessary abort statements.

VI FAILURE SEMANTICS FOR Ada

The remaining sections of this paper show how Ada can be used in a fault-tolerant distributed system. A first step in the construction of such a system is to fill in the gaps in the semantics of Ada mentioned in the previous section. In particular, the meaning of distribution and the effect of damage to the remaining software caused by a processor failure must be specified. While it is not difficult to choose a reasonable meaning for distribution, the problem of what to do with damaged tasks is much more difficult. It must be emphasized that the semantics suggested in this section were chosen so as to follow the existing language as much as possible.

Distribution

The primary aim of distribution semantics is to avoid the possibility of a centralized distributed system. It will be assumed that only tasks will be distributed. The distribution of a task T to a processor P will be taken to mean that the task activation record for T and all of the code for T will be resident on P.

Damaged Tasks

In section IV it was pointed out that the failure of a processor may affect tasks running on the remaining processors, and that many language features can cause these problems. The difficulties do not arise because tasks were lost when the processor failed. Any task could be removed from an Ada program at essentially any point _without_ processor failure by execution of an _abort_ statement. Rather, the difficulties arise because the semantics of the language fail to deal with this situation. Ada semantics _are_ precisely defined for tasks being aborted and for the subsequent effects on other tasks, and the execution-time system is required to cope with the situation. We suggest therefore that that damage following processor failure can be handled as if the task that were lost had been aborted. This would allow the language-dependent part of the damage following processor failure to be treated using existing language facilities.

This choice of semantics leads to a final problem that needs to be addressed; the status of the main program following failure. By definition all non-library tasks in an Ada program are nested inside the main program and so depend upon it. If failure of a processor is to be treated as if abort statements had been executed on the lost tasks, then a serious problem arises with the main program. When a task is aborted, all its dependents are aborted also. For any task lost through failure this is reasonable. It means that all the dependents that were not lost with the task have to be aborted by the system. However, if the main program is lost, this implies that all the tasks that depend on the main program (that is almost all of them) will have to be aborted. This effectively removes the entire program. Clearly this is unsatisfactory. We suggest therefore that the main program has to be treated as a special case. For the main program, and only for the main program, the execution-support system will have to create an exact replacement if the main program is lost through processor failure. To ease the overhead that this involves, we suggest that the main program be limited to a single null statement.

VII A SUPPORT SYSTEM STRUCTURE FOR Ada

Although Ada does not support the facilities required for continuation explicitly, the semantics described in section VI can be achieved if the execution-time support structure is suitably modified. In this section we discuss the necessary modifications. In section VIII we show how they are used with Ada.

Failure Detection

Failure detection could be performed by hardware facilities over and above those provided for normal system operation. Alternatively, failure could be detected by system software. The hardware option is less desirable because it requires additions to existing or planned systems, and the detection hardware itself could fail. We suggest therefore the use of software failure detection.

Software failure detection can be either passive or active. A passive system might rely on tasks assuming that failure had occurred if actions did not take place within a "reasonable" period of time i.e. timing out. Alternatively, a passive system could require that all messages passed between tasks on separate processors be routinely acknowledged. This is a particularly simple case of timing out since failure has to be assumed if no acknowledgement is received.

The disadvantages of passive detection are:

(1) Timing out assumes an agreed-upon upper limit for response time.

(2) A failed processor will not be detected until communication is attempted and this may be long after the failure has occurred.

Upper bounds on response time may be hard to determine. Very complex situations can arise from an incorrect choice. The reason for a lack of response from a task on another processor may not be failure of that processor but merely a temporary rise in its workload. The consequences of timing out could be an assumption by one processor that another had failed, followed by reconfiguration to cope with the loss. Clearly, if this assumption is wrong, two processors could begin trying to provide the _same_ service.

Being unaware that a processor had failed will lead to a loss of the service it was providing until the failure is noticed. In a system with many processors each providing relatively few services, the amount of inter-processor communication might be quite low. A failed processor may then go unnoticed for so long that damage to the system being controlled might result.

It is for these reasons that we reject passive software failure detection and suggest the use of active software failure detection. In an active system, some kind of inter-processor activity is required periodically and if it ceases, failure is assumed. The messages that are passed are usually referred to as heartbeats. Multiple failures may occur at essentially the same time but transmission times may vary. Since it is important that machines surviving failure have a consistent view of the system state before they begin reconfiguration, the heartbeats must be organized so that each remaining processor gets the same information about the failure. There are many ways to achieve this. For example, all machines may be required to generate their heartbeats at approximately the same time so that each machine will receive all the heartbeats of the other machines in a given interval. Any not received in this interval can be assumed to have failed.

A final question of implementation is whether the generation and monitoring of heartbeats should be the responsibility of the programmer or the Ada execution-time support system. We favor the execution-time support system for reasons discussed below.

Damage Assessment

The mechanism that we propose to cope with this, together with the heartbeat mechanism, is shown in figure 2.

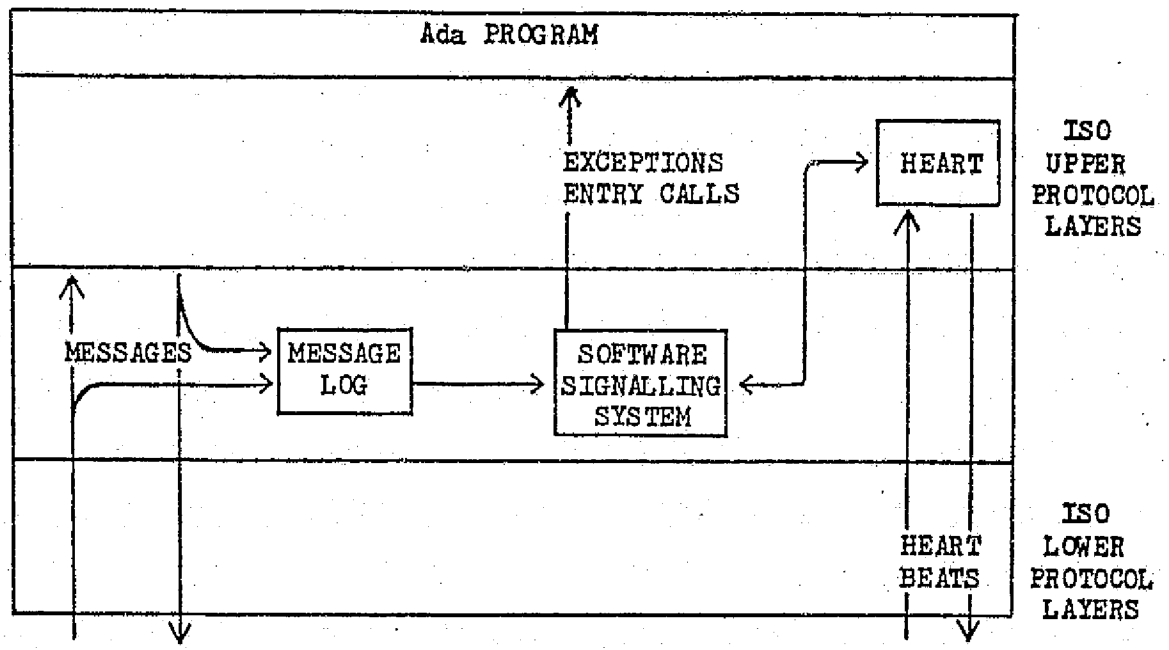


Figure 2 - Implementation Model

Whenever any communication takes place between tasks on different processors, the execution-time support system on the processor starting the communication records the details in a <u>message log</u>. Whenever a failure is detected, each processor checks its message log to see if any of its tasks would be damaged by the failure (permanently suspended for example). If any are found, they are sent fake messages. They are called "fake" because they are constructed to appear to come from the failed processor but clearly do not. The message content is usually equivalent to that which would be received if the lost task had been aborted. In this way, each processor is able to ensure that none of its tasks is permanently damaged, and the action following failure for each remaining task is that which is associated with an abort. It often takes the form of raising an exception.

Clearly it is possible for unsuspecting tasks to attempt to rendezvous with tasks on the failed processor after failure has been detected, signaled, and other communications terminated. This situation can be dealt with easily if the execution-time support system returns a fake message immediately indicating that the serving task has been aborted and that rendezvous is not possible.

Because of the fact that a fairly extensive set of facilities is required in the execution-time system for fake messages, we suggest that the heartbeats be handled here also. There is a clear need for cooperation between the heartbeat monitoring system and the fake message system. Operating both at the same level is probably the only practical approach. This has the additional advantage that the programmer is not burdened with the need to include the heartbeat system in his program. Finally, the heartbeat system is so central to the reliability of the entire system that it should operate at the lowest practical level of the software system. Thus it relies for its operation on the correct operation of the minimum amount of other software.

Selecting and Effecting The Response

Since consistent data across machines is essential to allow a response to be chosen and implemented, the execution-time support system for Ada must provide a mechanism for ensuring that data can be distributed reliably. As mentioned in section IV, a two phase protocol can be used and we propose that an implementation of it be included in the execution-time system along with the message log and heartbeat mechanism.

VIII FAULT TOLERANCE IN Ada

In this section we show how a fault-tolerant Ada system can be built using the support mechanisms just described. As was shown in section V, Ada does not provide any specific facilities to support this type of fault tolerance. Existing features of the language that were not designed for the purpose have to be used to interface with the modified support system.

Failure Detection

When failure of a processor is detected by the heartbeat mechanism, this information must be transmitted to the software running on each remaining processor so that reconfiguration can take place. The information is available to the execution-time support software in some internal format, but it has to be transmitted to the Ada software using an existing feature of the language.

As noted in section V, if the language is not to be changed one approach is to make use of the language's exception mechanism, and have the execution-time system generate a predefined exception on each processor remaining after a failure. If this is done, it is not clear where the handler for the exception should be placed. The handler will be receiving extremely important information (namely that a failure has occurred) and, in order to deal with the situation, it must be guaranteed that the handler will be executed. Unless handlers for the exception are placed in every task that might be running, execution of the handler cannot be guaranteed. An alternative is to define a special task to contain the handler and to raise the exception in this task only. Clearly, the special task should not be executing until the exception is raised. Unfortunately, there is no way to activate a task with an exception in Ada.

Another approach to signaling failure is to view the required signal as being very like an interrupt, and transmit the information to the Ada software by a call to a predefined entry. Again there is the problem of where the entry should be defined to ensure execution. However, in this case, the solution of defining a special task and defining the entry within it works very well. If the task is given a very high priority, it will be suspended on the entry until the call that signals failure, whereupon it will immediately begin execution.

We propose therefore that a special task (RECONFIGURATION_I) be defined on each processor (I being the processor number) that will contain a special entry with a single parameter. The _accept_ statement associated with the entry will be in an infinite loop. This task will normally be suspended on the _accept_ statement for special entry and, when a failure is detected by the heartbeat mechanism, a call to the entry will be generated. The parameter passed will designate which element of the system's hardware has failed. The task will then be activated and will contain code within the _accept_ statement to handle reconfiguration. A general form of the body of this task is shown in figure 3. Since this task is in an infinite loop, it returns to the _accept_ statement once a particular failure has been dealt with. Thus subsequent failures will be dealt with in the same way and, in principle, any number of sequential failures can be dealt with. Further, if physical damage removes more than one processor at the same

```
task body RECONFIGURATION_I is
begin

   -- initialization code
   loop
      accept FAILURE(WHICH : in failure_types) do
         -- code to handle hardware failures
      end accept;
   end loop;

end RECONFIGURATION_I;
```

Figure 3.

time, the remaining processors will notice the loss of heartbeats in some order, and calls to the entry in the reconfiguration task will be generated sequentially. Thus multiple failures occurring together will be dealt with as if they had occurred in some sequence.

Damage Assessment

Given the support system described in section VII, damage will be limited to lost tasks and lost data. No remaining tasks will be suspended. Each task that could have been suspended will have received fake messages giving it the impression that the task it was communicating with had been aborted. It is the programmer's responsibility to ensure that the subsequent actions by these tasks are appropriate. In addition, all tasks losing contexts will have been aborted.

The tasks and data that were lost need to be determined. Provided there is control of distribution (see below), this is quite simple. The information about which tasks are on which processors could be maintained and referenced in three ways:

(1) It could be stored in a table within the program itself.

(2) It could be stored within a table maintained by the execution-support system. This table will be needed in any case because it is required to implement inter-task communication. Provision could be made for the program to interrogate it.

(3) The information could be stored implicitly within the program. If all task-to-processor assignments are known at compile time and do not change, the code used for reconfiguration following failure can be written with the distribution information as an assumption.

There is no clear advantage to any of these methods. The choice in any particular case is implementation dependent. In the example given in appendix 1 we use the third method.

Selecting and Effecting The Response

Algorithms for the selection of a suitable response, and the algorithms used in that response, depend for their correct operation on having appropriate data available. Each piece of data being manipulated by a program for a typical embedded application can be regarded as either expendable or essential. Partial computations and sensor readings are expendable whereas navigation information or weapons' status are essential.

Expendable data need not be preserved across machine failures. A partial computation, for example, is only of value to the expression generating it. Replacement software that will be used following a failure can simply recompute any expendable values. A sensor value for example, is usually only useful for a short time and a suitable replacement value can be obtained by reading the sensor again. We suggest that any data items that the programmer considers expendable be given no special attention, and that the replacement software be written with the assumption that these data items are not available.

Essential data does need to be maintained across machine failures. In an Ada program this could be implemented in two ways. First, data items that the programmer considered to be essential could be marked as such (perhaps by a pragma), and the system would then be required to ensure that copies of these data items were maintained on all machines. Each time the data item was modified, all the copies would be updated. In the event of failure, one of the backup copies could be used immediately. This is simple for the programmer but potentially inefficient. Consider for example a large array that was designated as essential If it were being updated in a loop, as each element was changed, it would be necessary to update all the copies of that element. The entire overhead associated with maintaining consistent copies would be incurred for each element change. In practice in order to allow reconfiguration, it would probably be adequate to wait until the all the elements of the array had been modified and then update all copies of the array at once.

A second approach is to provide the programmer with the tools to generate consistent copies across machines. In this way, not only the data items to be preserved but also the times during execution when copies will be made will be under the programmer's control. We suggest that this could be done by providing a special task (DATA_CONSISTENCY_I) on each processor that will contain an entry with a single parameter. The parameter would be a record with a number of variant parts, one part for each essential data item. Calling the entry and passing the latest value of a data item in the record together with the appropriate discriminant causes the necessary copies to be made and distributed while the calling task waits. Completion of the rendezvous indicates that this process has been satisfactorily completed. A general form of the body of this task is shown in figure 4.

Another interface that could be used to provide access to the data consistency support facilities is a generic package. A generic package could be defined containing a task or procedure that can be called to distribute values of essential variables. The generic parameter for the package would be the type of the essential variable and a new instance of the package would be instantiated for each essential data item in the program. The number of packages can be kept low by grouping sets of essential data items into records.

Although there are no specific facilities for control of task distribution, limited control can be achieved using either an implementation-dependent pragma or an implementation-dependent address clause. The pragma could have a machine notation as a parameter and be

```
task body DATA_CONSISTENCY_I is
begin
   loop
      accept DISTRIBUTE_VALUES(DATA_VALUE : in DATA_TYPE) do
         -- implementation of two phase protocol
      end accept;
   end loop;
end DATA_CONSISTENCY_I;
```

Figure 4.

required to appear in the specification of the task or task type to which it applies, as is done with the predefined pragma priority. This pragma would require the compiler to generate instructions and loader directives for the designated task to ensure that it is placed in the required machine. This is the notation used in the example given in appendix 1. Alternatively, for tasks created by allocators, the pragma could be required to appear in a declarative part and it would apply only to the block or body enclosing the declarative part (similar to the predefined pragma optimize). Its effect would be to cause all tasks created by allocators in the body or block to be distributed to the machine designated by the pragma. Similarly, for a particular implementation, the identifier parameter in the address clause could be interpreted as a task name, and the expression parameter as a machine designation.

As was pointed out in section V, the creation and deletion of tasks that might be required as part of effecting a response is easily achieved in Ada using allocators and the abort statement. Thus the particular accept statement within the reconfiguration task that is executed for a given failure can create and delete whatever tasks are needed to provide alternate service. A simpler approach to providing replacement software is to arrange for the required replacement task to be present and executing before the failure, but suspended on an entry. Such a task would not consume any processing resources although it would use memory, but it could be started by the reconfiguration software very quickly and easily by calling the entry upon which the replacement task is suspended. A general form for a replacement task is shown in figure

5. This is the mechanism used in the example in appendix 1.

Redirection of communication to alternate software that has been started following a failure has to be programmed ahead of time into tasks that call entries in tasks that might fail. It will be necessary for the reconfiguration task on each processor to make status information about the system available to all tasks on that processor. Each task must then interrogate this information before making a call to an entry on a remote machine in case the entry has changed because of failure.

In summary, an Ada program that uses the support system described in section VII to allow it to tolerate the loss of one or more processors would have the following form:

(1) A main program consisting of a single null statement.

```
task ALTERNATE SERVICE is
   pragma distribute(PROCESSOR_I);
end ALTERNATE SERVICE;

task body ALTERNATE_SERVICE is
begin
   -- Code necessary to initialize this alternate service.
   accept ABNORMAL_START;
      -- This task will be suspended on this entry until it is
      -- called by RECONFIGURATION_I following failure.  The
      -- code following the accept statement provides the
      -- alternate service.
end ALTERNATE_SERVICE;
```

Figure 5.

(2) A static structure in which there is little or no nesting of the application and alternate tasks themselves. They may define nested tasks for their own use. This is necessary to ensure that these tasks are visible to the reconfiguration and data consistency tasks.

(3) A set of tasks providing the various application services; the distribution of the tasks being controlled by an implementation-defined pragma or address clause. Each task would contain handlers for exceptions (such as <u>tasking error</u>) that might be generated by the support system if failure occurred while that task was engaged in communication with a task on a remote machine.

(4) A set of tasks designed to provide any alternate service that the programmer chooses; each alternate task suspended on an <u>accept</u> statement that will be called to start it executing.

(5) A task on each processor designed to cope with reconfiguration on that processor; this task containing one entry for each hardware component that might fail. These entries would be called automatically be the support system following failure detection.

(6) A task on each processor designed to distribute copies of essential data for tasks on that processor. Rendezvous with this task allows any other task to distribute essential data at any time the programmer chooses.

IX CONCLUSION

Although the probability of failure per unit time for a modern fault-tolerant processor is low, it is not zero. The loss of processors in a distributed system is certain to occur and must be anticipated. In order to benefit from the flexibility of distributed processing, crucial systems must be able to deal with processor failures.

Ada was designed for the programming of embedded systems, many of which are crucial and distributed. We have examined Ada's suitability for programming distributed systems in which processor failure has to be tolerated and found it to be inadequate. The difficulties have been discussed and proposals to avoid them have been suggested. These proposals involve extensive modification to the execution-time system used by Ada and careful organization of the Ada program itself but no language changes.

Although the discussion presented here is in terms of distributed systems, similar problems can arise in shared-memory multiprocessor systems where processor failure has to be anticipated. If the system is organized so that different processors execute different tasks, processor failure at an arbitrary point could produce exactly the same damage as was discussed in section V.

We consider the non-transparent approach to be the only one that is feasible and this requires language facilities for its support. The fact that Ada makes no explicit provision for this type of fault tolerance is unfortunate. The solution presented in this paper uses existing features of the language and is far from ideal. Modifications to Ada would be preferable.

An implementation of the tasking and exception handling features of Ada incorporating the ideas described in this paper is being undertaken. It is presently being tested.

Appendix 1 of APPENDIX 5

A Programming Example

This is a very simple example designed to illustrate some of the ideas discussed in this paper. In a typical Ada application, the program would be much larger and would have to take into account all the language features mentioned.

The example consists of a calling task CALLER that operates on one processor (processor one) and a serving task SERVER that operates on another processor (processor two). The calling task does some real-time processing and calls an entry in the serving task in order to get some kind of service. The program is written to cope with failure of either processor. Alternates are provided for the calling and the serving tasks and a reconfiguration task is present on each processor.

Normally only the calling and serving tasks are executing and a fault-intolerant version of this example would consist of just these two tasks. If processor one fails then it is necessary to start an alternate calling task on processor two. Similarly, if processor two fails it is necessary to start an alternate serving task on processor one.

The alternates are present on the required machines when the program starts execution. Each alternate is waiting on an entry named ABNORMAL_START so that they do no processing while both processors are operational. When one processor fails, the run-time system generates an entry call on the other processor to an entry in its task RECONFIGURE_I (where I is the processor number). This task then calls the ABNORMAL_START entry for the alternate that is needed and processing is able to continue. Entries are defined in RECONFIGURE_I for each component that might fail. In this example, each machine is only interested in the failure of the other so only one entry is defined in each reconfiguration task.

If a rendezvous is in progress when the failure occurs, then the serving task need not care that the calling task has been lost, and the rendezvous can complete. The calling task will care if the serving task has been lost because this will indefinitely suspend the caller. Thus TASKING_ERROR is raised by the run-time system in the calling task. This frees the calling task and allows it to prepare itself to use the alternate server.

Note that the server does not need to be aware that the caller has been replaced by an alternate if the caller's machine fails because the rendezvous is asymmetric. The entries in the server can be called by any task; in particular both the caller and the alternate caller.

If a rendezvous is not in progress when the failure occurs then processing on the remaining processor can continue. If this processor is executing the caller, then the caller will receive TASKING_ERROR the next time it attempts to rendezvous with the server and it will be reconfigured at that time. The alternate server will have already been started by then.

In this example, a single array called DATA is regarded as essential and the programmer has decided that it needs to be made consistent across both machines only rarely. The entry DISTRIBUTE_VALUES is called in tasks DATA_CONSISTENCY_1 and DATA_CONSISTENCY_2 periodically to achieve this.

Distribution is controlled in this example by the pragma DISTRIBUTE that takes a processor number as its parameter. It must appear in the specification of the task to which it applies. It is much like the predefined pragma PRIORITY.

-- Text For The Example Will Be Added Later --

Appendix 2 of APPENDIX 5

Damage In Task Communication

In this section, the simple rendezvous will be examined in detail as an example. Other language elements involving task communication are considered only briefly since, for the most part, the difficulties that arise from processor failure are similar to those that arise in the simple rendezvous. The phrase "the processor executing task X fails" will be abbreviated to "task X fails" whenever no confusion arises.

A simple rendezvous in Ada consists of a calling task C making an entry call, S.E, to a serving task S, that contains an accept statement for the entry E. The syntax is shown informally in figure 6. The semantics of the language require that if the call is made by C before the accept statement is reached by S, C is suspended until the accept statement is reached. If S reaches the accept statement before the call is made by C, S is suspended until the call is made. In either case, C remains suspended until the rendezvous itself is complete.

```
        Calling Task C                        Serving Task S

              .                                  ACCEPT E DO
              .                                       .
              S.E;                                    .
              .                                       .
              .                                  END E;
```

Figure 6 - The Syntax Of A Simple Rendezvous.

In order to look at the effects of processor failure on the rendezvous, it is necessary to specify an implementation at the message passing level. Only the simple case of a task C calling an entry E in a serving task S will be considered. Further, we assume that the call is made before S has reached the corresponding accept statement; the case where the serving task waits at its accept statement is similar. One possible message sequence is shown in figure 7.

The calling task C asks to be put onto the queue for entry E. When S reaches its accept statement for E, it sees that C is on the queue. C can be considered to be engaged in the rendezvous after the RENDEZVOUS_START message arrives at C. When the rendezvous is completed the RENDEZVOUS_COMPLETED message would awaken C which would continue. We assume that all messages arrive safely.

Using this implementation of a simple entry call, what happens if either the serving or the calling task fails? A detailed examination of all possible cases has appeared elsewhere [13] and will not be repeated

```
      Calling Task C                              Serving Task S

          S.E;
                      PUT_ONTO_QUEUE---------->
    |                 <---------REQUEST_QUEUED
   Time                                            ACCEPT E DO
    |                 <-------RENDEZVOUS_START
    |                                                   .
    V                                                   .
                                                   END E;

                      <----RENDEZVOUS_COMPLETE
```

Figure 7 - The Messages Used To Implement The Rendezvous

here. However, it is clear that there are several situations in which processor failure could cause one of the tasks to be trapped. For example, if task S fails at any point after the RENDEZVOUS_START message is sent but before the RENDEZVOUS_COMPLETE message is sent, the latter will never be sent. Task C has no way of distinguishing this situation from a long service time by task S, and so will wait forever. Although the processor executing C is still working, task C is permanently suspended by the loss of a different processor.

It might appear that the timed entry call solves some of the problems raised above but it does not. The semantics of the timed entry call appear to be quite straight-forward [2]:

> A timed entry call issues an entry call that is canceled if a rendezvous is not started within a given delay.

However, in a distributed system, messages will take time to get from a task on one processor to a task on another. Even if the underlying message passing system can guarantee that a message will eventually arrive correctly, this will be implemented at a lower level by a protocol that may well involve acknowledgement of messages, and the resending of messages that have been lost. A message can certainly be delayed for some arbitrary length of time. Even physical separation of the processors may impose a significant delay.

One possible interpretation of the timed entry call would be to count the total time until the rendezvous is started. Message passing time and time waiting on the entry queue would be included. This interpretation has to be ruled out by the statement in the language definition that a timed entry call with a delay of zero is the same as a conditional entry call:

> If a rendezvous can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed and the optional sequence of statements after the call is then executed.

If the delay included both message passing time and time on the queue, a delay of zero would be impossible and a timed entry call with a delay of. zero would never succeed.

Another interpretation of the delay is that it is just the time spent waiting on the entry queue. We assume that this is the delay intended by the language definition since this has a meaning when the specified delay is zero.

A timed entry call gives protection against having to wait too long on the entry queue. Thus, it could be used to provide protection against processor failure <u>before</u> the rendezvous starts but not afterwards. An analysis of the message traffic necessary for the timed entry call can be performed that is similar to that shown in figure 7. The issues that arise when considering failure are similar but more extensive than those that arise with the simple rendezvous. What the task issuing the call needs is some guarantee that it will not be trapped in an attempt to communicate, and forced to miss a deadline. It does not matter to the task, whether the time is spent waiting on a queue, or attempting to send a message, or any other activity.

The conditional entry call is no more helpful than the timed entry call. Again, the semantics of the conditional entry call appear to be quite straight-forward [2]:

> A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible.

By a similar argument to that used with timed entry calls, we conclude from the rules of the language that "immediately" must mean zero waiting time on the entry queue. As message passing time can vary, "immediately" may turn out to be an arbitrary delay. Apart from the semantic difficulties arising in a distributed system, the possibility of the caller being trapped indefinitely following processor failure occurs with conditional entry calls as with the other rendezvous.

We now consider the creation of nested tasks. Again, a detailed examination of the difficulties arising with task creation has appeared elsewhere [13]. Here, we give an example to show the potential problems. Task creation by allocators will not be considered; the difficulties that arise are similar.

A task is created in two steps. First, it is elaborated at which point entry calls can be made to it. Second, it is activated, that is, the declarative part of its body is elaborated and it begins execution. Elaboration of a task occurs as a part of the elaboration of the body of the declarative part of the parent unit. Activation of a task occurs after the elaboration of the declarative part of the parent unit. Conceptually, this occurs after the <u>begin</u> but before the first statement of the parent's body. The parent cannot be activated until all nested tasks have been activated.

To see the difficulties that task creation can raise, consider a set of three tasks P, A, and B, with A and B nested inside P. A and B have no other tasks nested within them and each task is to execute on a different processor; P on processor one, A on processor two, and B on processor three. The elaboration of the body of P includes the elaboration of A and B, and so messages will be sent from processor one to the other processors requesting the elaboration of A and B. Once this is done, tasks A and B can accept entry calls. When task P reaches its begin, all of the objects that it declares have been elaborated, and A and B are then activated. This requires an "activate A" message being sent from processor one to processor two and an "activate B" message being sent from processor one to processor three. The activation of P requires that the activation of A and B be complete, and so P cannot proceed until it has received responses to these activation messages indicating that A and B have been activated. Clearly there are numerous difficulties that can arise if any one of the three processors fails. For example, P will be suspended forever if either processor two or three fails at any time before both A and B have completed activation. In that case, P could not proceed because one of its dependents would never be activated. Similarly, A would be trapped if it called an entry in B and processor three failed after B was elaborated but before it was activated. Both A and B would be trapped if processor one failed after the elaboration messages were sent to A and B, but before the activation messages were sent.

Task termination produces difficulties also. A task waiting at a select statement with an open terminate arm can be terminated if its master is completed and all other dependents of the master are either terminated or waiting at _select_ statements with open terminate arms. In order to check this condition it is necessary to suspend all the dependents as they are checked. If this is not done, it is possible for two dependents each to be waiting at a _select_ statement with an open terminate when checked, but never to both be waiting at _select_ statements simultaneously. If the dependents are executing on a different processor from the master, it is necessary for the master to send messages to its dependents suspending them for the duration of the termination check. If the master task's processor fails before a message to resume is sent (assuming that termination is not possible), a suspended dependent will remain suspended forever.

Even accessing a non-local or a shared variable could cause the referencing task to be suspended. Access to a variable stored on a remote machine requires that a request message be sent and that a reply be received. Failure of the remote machine between the two messages would cause suspension of the requesting task. Although the language allows an implementation to use a copy of a shared variable, updating it only at synchronization points, and the definition of a synchronization point can vary depending on whether or not the variable is declared as _shared_, there are still implied update messages at synchronization points. Access to a shared variable on another processor requires that a dialogue take place, and failure of the processor on which the shared variable resides could trap the task attempting to reference the variable.

REFERENCES

(1) McTigue, T. V., "F/A-18 Software Development - A Case Study", Proceedings Of The AGARD Conference On Software For Avionics, The Hague, Netherlands, September 1982.

(2) Reference Manual For The Ada Programming Language, U. S. Department of Defense, 1983.

(3) Department Of Defense Requirements For High-Order Computer Programming Languages - STEELMAN, U. S. Department of Defense, 1978.

(4) Wensley, J. H. et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(5) Hopkins, A. L., et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

(6) Tanenbaum, A. S., "Network Protocols", ACM Computing Surveys, Vol. 13, No. 4, December 1981.

(7) Schlichting R. D., and F. B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", ACM Transactions On Computer Systems. Vol. 1, No. 3, 1983.

(8) Cornhill, D., "A Survivable Distributed Computing System For Embedded Applications Programs Written In Ada", ACM Ada Letters, Vol. 3, No. 3, December 1983.

(9) Leveson, N. G., and P. R. Harvey, "Analyzing Software Safety", IEEE Transactions On Software Engineering, Vol. SE-9, No. 5, 1983.

(10) Alsberg, P. A., and J. D. Day, "A Principle For Resilient Sharing Of Distributed Resources", Proceedings Of The International Conference On Software Engineering, San Francisco, October 1976.

(11) Gray, J. N., "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, New York 1978.

(12) Lee, P. A., "A Reconsideration Of The Recovery Block Scheme", Computer Journal, Vol. 21, No. 4, 1978.

(13) Reynolds P.F., J.C. Knight, J.I.A. Urquhart, "The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements", Final Report on NASA Grant No. NAG-1-260, NASA Langley Research Center, Hampton, Va.